Management and Evolution of Business Process Variants

Pavel Bulanov

This research was supported by the Netherlands Organization for Scientific Research (NWO) under project number 638.001.207 whithin the scope of the Jacquard program.

RIJKSUNIVERSITEIT GRONINGEN

Management and Evolution of Business Process Variants

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. E. Sterken,
in het openbaar te verdedigen op
dinsdag 11 december 2012
om 11.00 uur

door

Pavel Bulanov

geboren op 2 augustus 1980
te Bobruysk, Wit-Rusland

Promotor:	Prof. dr. M. Aiello
Copromotor:	Dr. A. Lazovik
Beoordelingscommissie:	Prof. dr. P.W.P.J. Grefen Pror. dr. M. Aksit Prof. dr. M. Mecella

ISBN: 978-90-367-5864-2
ISBN ELECTRONIC VERSION: 978-90-367-5966-3

Contents

Acknowledgements **v**

Samenvatting **vii**

1 Introduction **1**
1.1 Case–Study: Variability in Local eGovernment 4
1.1.1 Business Process Variability 5
1.1.2 Business Process Evolution 5
1.1.3 Runtime reconfiguration 7
1.2 Management of Business Process Variants 9

2 Related Work **13**
2.1 Business Process Variability 14
2.1.1 Declarative vs. imperative variability 15
2.1.2 Annotation–based Variability 18
2.1.3 Variability by Underspecification 22
2.1.4 Declarative Process Execution 24
2.1.5 Runtime Business Process Reconfiguration 27
2.2 Business Process Generation 28
2.2.1 Process Merge 28
2.2.2 Workflow Mining 33
2.2.3 Other Approaches to Business Process Generation 35

3 The Case of Design Time **39**
3.1 Basic Definitions 41
3.2 Template Design 43

3.2.1 Declarative Techniques . . . 44
3.2.2 Imperative Techniques . . . 50
3.3 Constraint Relations . . . 56
3.3.1 Prerequisite . . . 56
3.3.2 Exclusion . . . 57
3.3.3 Substitution . . . 57
3.3.4 Corequisite . . . 57
3.3.5 Exclusive–Choice . . . 58
3.4 Variant Design: An Example . . . 58
3.5 Process Healthiness . . . 61
3.6 Variant Validation . . . 61
3.6.1 Model Conversion . . . 62
3.6.2 Validation Algorithm . . . 62
3.7 Evaluation . . . 66
3.7.1 The Imperative Case . . . 66
3.7.2 The Declarative PVDI Case . . . 67
3.7.3 Expressive Power . . . 67
3.7.4 Ease of Use . . . 69
3.8 Implementation and Performance Results . . . 74
3.9 Discussion . . . 75

4 Business Process Transformation **79**
4.1 A Temporal Logic of Business Processes . . . 80
4.1.1 Temporal Process Logic . . . 81
4.1.2 Discussion . . . 85
4.2 Process representation and transformation . . . 86
4.2.1 Transformations . . . 91
4.2.2 Business Process Evolution through Transformation Functions 96
4.2.3 Implementation and Evaluation . . . 97
4.3 Discussion . . . 97

5 The Case of Run Time **101**
5.1 Runtime Variability using Dependency Scopes . . . 102
5.1.1 Dependency Scopes within WMO Process Example . . . 103
5.1.2 Required Intervention Processes . . . 103
5.1.3 Automatic Intervention Process Generation . . . 106
5.2 Architectural Overview . . . 107
5.3 Basic Concepts . . . 109
5.3.1 Business Process . . . 109

5.3.2 Dependency scope . . . 112
5.3.3 The Planning Domain . . . 115
5.4 Automatic Intervention Process Generation . . . 116
5.4.1 Generation of the Planning Domain . . . 116
5.4.2 Composition of the initial planning state . . . 118
5.4.3 Generating the IP . . . 118
5.5 The Prototype . . . 119
5.5.1 Process Modeller . . . 120
5.5.2 The Process Executor . . . 121
5.5.3 The planner . . . 124
5.6 Evaluation . . . 124
5.7 Discussion . . . 125

6 Conclusion **129**

A Modal Logics in a Nutshell **133**
A.1 Linear Temporal Logic (LTL) . . . 134
A.2 Computational Tree Logic (CTL) . . . 135

Bibliography **137**

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Dr. Marco Aiello. During the whole study he has been encouraging me to dare the challenges of computer science and not hesitate to try another way or look from a different perspective on the same problem. Thanks to his razor-sharp feedback, I was constantly in a proper shape and ready to describe and defend the results of my work with my bare hands. It remains a mystery for me how did he manage to find time to deal with my questions and read the numerous revisions of my writings, even being overwhelmingly busy with other projects and occupied by a dozen of knowledge–lusting students.

In the same spirit, I thank my co-supervisor, Dr. Alexander Lazovik, who showed me that computer science is a bit more than typing Java code in Notepad. On those not so infrequent occasions when Marco was quite busy, Alexander was always ready to take the lead and to discuss any new ideas, no matter how crazy might they look at a glance. Yet it did not stop him from telling the bitter truth should he see a weak point in the middle of an ingenious proof.

I thank the members of the reading committee Prof. Dr. Massimo Mecella, Prof. Dr. Mehmet Aksit, and Prof. Dr. Paul Grefen for careful examination of my thesis. In addition, I would like to thank Prof. Dr. Wim Hesselink, who thoroughly revised the formal part of the first two-thirds of this thesis.

I acknowledge the importance of cooperation and joint work with my colleagues and members of SaS-LeG team, and I specially note Heerko Groefsema, my inevitable co-author and just a kind person who agreed to translate the summary of this thesis into Dutch. I also thank Nick van Beest and Eirini Kaldeli, with whom I spent countless hours discussing the different aspects of process variability and who were also noticed in co-authoring with me. I thank Prof. Dr. Hans Wortmann for his valuable feedback, Prof. Dr. Paris Avgeriou for the occasional inspirations during my study, Dan Tofan for trying to make a common research despite the fact that we never actually made it, and Dr. Matthias Galster for showing an example of a productive researcher and fast writer.

I would also like to thank my colleagues from the Distributed Systems research group who are at the same time competitors for a precious and limited resource: time slots for meeting with Marco. Still, we have nice and friendly atmosphere in our group, and here are the people responsible for that, in random order: Saleem Anwar, Faris Nizamic, Ehsan Ullah Warriach, Viktoriya Degeler, Ilche Georgievski, Ando Emerencia, Andrea Pagani, and Tuan Anh Nguyen. I would also like to mention former colleagues and passers–by from the department, some of whom also managed to graduate in the last couple of years and thus inspired me with positive examples. Here they are: Dr. Aree Witoelar, Dr. Kerstin Bunte, Dr. Petra Schneider, Elena Lazovik, Alexander Bograd, George Azzopardi, Ioannis Giotis, Mahir Can Doganay, Elie El-Khoury, and Artemios Kontogogos.

It goes without saying that my family provided me with the best support and encouragement, even being half a continent away. My mother wanted me to travel around the world and stay home at the same time, which posed a challenge I had to deal with somehow. Yet she actively encouraged me to start a PhD study and to cope with the troubles in the middle, and it would not be possible to complete this thesis without her constant support and patient waiting. My father always encouraged my to reason in a scientific way and always to justify my yet unripe conclusions on the nature of things. Thanks to him, I was immune to the call of fashion and chose the way of computer science neglecting all the fancy trends. I deeply regret that he did not live to see this thesis printed, but I am sure he would have been happy and proud of me.

Finally and most specially, I thank my wife Olga for all the patience and support she demonstrated over the years of waiting and wondering. We are together now in spite of all the destructive efforts, which instigates me to ponder on the truth of the following quotation: “Whatever may befall thee, it was preordained for thee from everlasting” [Augustus 167 A.D.].

Pavel Bulanov
Groningen
November 14, 2012

Samenvatting

Bedrijfsprocesmanagementsystemen (BPMS) maken het mogelijk om processen binnen bedrijven en organisaties te automatiseren, en vertegenwoordigen voor deze entiteiten doorgaans de centrale doelstelling en toegevoegde waarde. Een grote hoeveelheid onderzoek is besteed aan het voorzien van intuïtieve en krachtige modelleerhulpmiddelen en infrastructuren ter ondersteuning van het uitvoeren van bedrijfsprocessen. Met de uitgestrekte adoptie van deze hulpmiddelen en de groeiende complexiteit van organisaties, is er echter meer ondersteuning nodig. Wanneer een organisatie groot genoeg is om uit afdelingen en gelieerde ondernemingen te bestaan, zou een situatie kunnen ontstaan waar afzonderlijke afdelingen vergelijkbare processen hebben geïmplementeerd op soortgelijke maar niet exact dezelfde manier. Een dergelijke familie van soortgelijke processen kan gezien worden als een reeks verschillende versies van hetzelfde generieke proces, maar blijven verschillende entiteiten vanuit het perspectief van de BPMS. Hierdoor wordt het moeilijk om alle versies tegelijk te onderhouden of om massamodificaties toe te passen wanneer bepaalde organisatiebrede regels veranderd zijn en deze veranderingen in veel bedrijfsprocessen tot uiting komen. Een verandering in wetgeving zou bijvoorbeeld veranderingen in alle gerelateerde bedrijfsprocessen kunnen impliceren.

In dit proefschrift wordt het probleem van het onderhouden van een reeks soortgelijke bedrijfsprocesversies geanalyseerd vanuit verschillende perspectieven. Eerst wordt een raamwerk met declaratieve basis voor het modelleren van generieke processen gepresenteerd. Met dit raamwerk bieden we een intuïtieve manier om bedrijfsprocessen, of families van bedrijfsprocessen, te beschrijven door middel van onze nieuwe visuele modelleertaal die op Business Process Modeling Notation (BPMN) gebaseerd is en is aangevuld met extra modelleerelementen. Deze extra elementen variëren van simpele eenrichtingspijlen die een relatie tussen twee bedrijfsproceselementen beschrijven tot gecompliceerde groepselementen die gehele sub-processen

omvatten. Voor elk van de visuele elementen is er een transformatieregel die voorschrijft hoe dat element in een reeks tijdslogicaformules wordt geconverteerd. Dit maakt het mogelijk om een visuele modelleeromgeving te hebben die uitbreidbaar is in de wijze waarop extra elementen toegevoegd kunnen worden zonder dat de gehele architectuur schade wordt betrokken.

Daarna wordt het probleem van het doorzetten van veranderingen behandeld in de vorm van geautomatiseerde bedrijfsprocestransformaties. Deze transformaties zijn gebaseerd op het vergelijken van twee bedrijfsprocessen op basis van hun temporale representaties. Als gevolg is de taak van het vergelijken van procesmodellen gereduceerd tot de taak van het vergelijken van twee reeksen van formules. Tevens utiliseren we, in plaats van bestaande tijdslogica, een nieuwe extensie die het mogelijk maakt om het verschil tussen de begrippen "altijd gevolgd door" en "soms gevolgd door" te onderscheiden. Dergelijk onderscheidend vermogen wordt bereikt door het gebruik van verschillende types vertakkingen in een bedrijfsprocesmodel die het mogelijk maken om rekening te houden met niet alleen de analyse van de paden in een graaf (zoals vele modale logica doen), maar ook de vertakkingspunten die zich in het pad bevinden samen met hun type (EN-splitsing of OF-splitsing). Met behulp van deze logica (welke we TPL noemen, Temporal Process Logic) hebben we een representatie die meer kan uitdrukken dan degenen die gebaseerd zijn op klassieke tijdslogica.

En ten slotte wordt het probleem van het dynamisch repareren van processen behandeld met behulp van eenmalig automatisch gegenereerde bedrijfsprocessen. Het feit dat een instantie van een bedrijfsproces incorrect is en gerepareerd moet worden wordt geïdentificeerd door middel van speciale bewakingsregels die samen met het bedrijfsproces gespecificeerd worden. In het geval van een verandering in de uitvoeringsomgeving, zoals een modificatie in de data, worden de bewakingsregels geverifieerd. In het geval dat één van deze regels overtreden wordt, moet tijdens de uitvoering een reparatieproces gegenereerd worden met behulp van geautomatiseerde planningstechnieken. De gehele lifecycle is ontwikkeld in de vorm van een prototype voor een bedrijfsprocesexecutiesysteem die het luisteren naar data modificatie gebeurtenissen, het verifiëren van bewakingsregels en het aanroepen van een Kunstmatige Intelligentie (KI) planner om een reparatieproces samen te stellen ondersteunt.

Samenvattend: Het raamwerk beschreven in dit proefschrift ondersteunt bedrijfsprocesvariabiliteit tijdens zowel de ontwerp- als de uitvoeringsfase met aanvullende ondersteuning voor bedrijfsprocesevolutie. Het denkbeeld is echter hetzelfde voor alle facetten van bedrijfsprocesmanagement en is gebaseerd op de representatie van een bedrijfsproces als een reeks logicaformules die daarna verwerkt en geanalyseerd worden met behulp van bestaande hulpmiddelen.

Het onderzoek dat in dit proefschrift gepresenteerd wordt is gedaan in samenwerking met het SaS-LeG project dat het doel heeft de applicatie van het Software als Services paradigma te analyseren voor het specifieke geval van de Nederlandse elektronische overheid. De in dit proefschrift geanalyseerde casussen waren daadwerkelijk verkregen ten gevolge van een aantal interviews verricht als deel van het SaS-LeG project. Als concrete casus beschouwen we de WMO–wet, aangezien deze in de praktijk veel moeilijkheden geeft door zijn complicaties en grote verschillen in de verwezenlijking door verschillende gemeentes.

Chapter 1

Introduction

Business processes are widely used to specify the lifecycle of an organization or a governmental institution. In a nutshell, a business process is a formal description of one specific routine task (or a series of routine tasks). More specifically, such a description comprises a set of actions, which must be executed in a certain order to fulfill a given task. Additionally, a business process can contain conditions which prescribe which actions must or must not be executed, depending on the current situation and the output from the actions already been executed.

Given a formal description of a business process, one can automate that process in such a way that, once each of the actions is implemented as a piece of software, a special Business Process Management (BPM) system can take care of the proper execution of the actions of the business process. Even if a certain action requires human decision, from the BPM point of view, such action is just a piece of software which has a user interface and whose invocation can take a long time, ranging from minutes to months. Consequently, a formal description of a business process is enough to execute that process in automated way, provided that the description is understandable by the business process engine in use.

Nowadays, specifically designed visual languages are used in order to describe a business process. One of these languages is called Business Process Modeling Notation (BPMN, [Object Management Group (OMG) 2009]) which has been released by the Object Management Group. Other examples of business process modeling languages include UML activity diagrams [(OMG) 2005] and Event–Driven Process Chains (EPC, [Sarshar and Loos 2005]).

There is extensive tool support by commercial vendors for business modeling that uses one of the languages mentioned above, such vendors as SAP, Oracle, Microsoft, and many other less well known. The positive result of using a visual business process modeling language is that such a model is easily comprehensible to a human user, and as a result there is a higher perceived level of abstraction.

However, such approaches do not include any explicit support for flexibility and change management, mainly because of the nature of modern business process modeling languages which are focused on fully defined and stable business processes; as a result, business process modeling tools *"enforce unnecessary constraints*

on the process logic" [van der Aalst, ter Hofstede and Weske 2003].

Variability in software engineering is defined as the ability of a software system to change and to support multiple versions at the same time. Such versions are typically called *variants* to emphasize the fact that those versions do not appear as the result of evolution but rather represent different customizations which may be used at the same time by different users. The initial version is usually called a *reference* or a *template* to reflect the fact that this version is used as a basis for later customizations.

The problem of variability is considered to be an important issue in the field of software product lines [Sinnema, Deelstra, Nijhuis and Bosch 2006, Pohl, Böckle and Linden 2005]. In the case of BPM, the problem becomes even more challenging, since the industrial business process modeling languages are oriented in terms of simplicity and ease of use and, as a result, lack the powerful features of general–purpose programming languages, such as class inheritance in object–oriented programming.

Moreover, different variants of the same business process constitute different entities from the Business Process Management system perspective, and a modification of one of these variants does not affect any of the others. On the contrary, it is often desirable to manage all the variants as a linked family, when: a change at the top level is propagated through the links to the variants, and, additionally, any change in a variant is verified for consistency with the rest of the family of business process variants. Specific problems of BPM systems require special techniques harmonized with visual business process modeling languages and the details of business process engines.

In [Balko, ter Hofstede, Barros and La Rosa 2009], the authors define, among the other things, the following research challenges in the field of business process variability: (i) **Reference Process Conformance** and (ii) **Reference Process Patchability**. The first challenge refers to the ability to validate a variant process and to establish whether it respects the important characteristics of the reference process. The second challenge refers to the ability to amend the reference process and to propagate the changes to its variants automatically or semi–automatically.

In this thesis, we will show how to overcome those challenges with the help of temporal logic formalisms, including the extension of well–known temporal logics which better suits our needs. Additionally, we will introduce a new approach to represent a business process as a set of logical formulas, which precisely describe a business process model while at the same time leaving room for later customization.

We will introduce the results of our research with the help of a case study taken from the field of Dutch e–Government. In the next section, this case study is described in detail, with the focus on the problem of variability in the case of business process management.

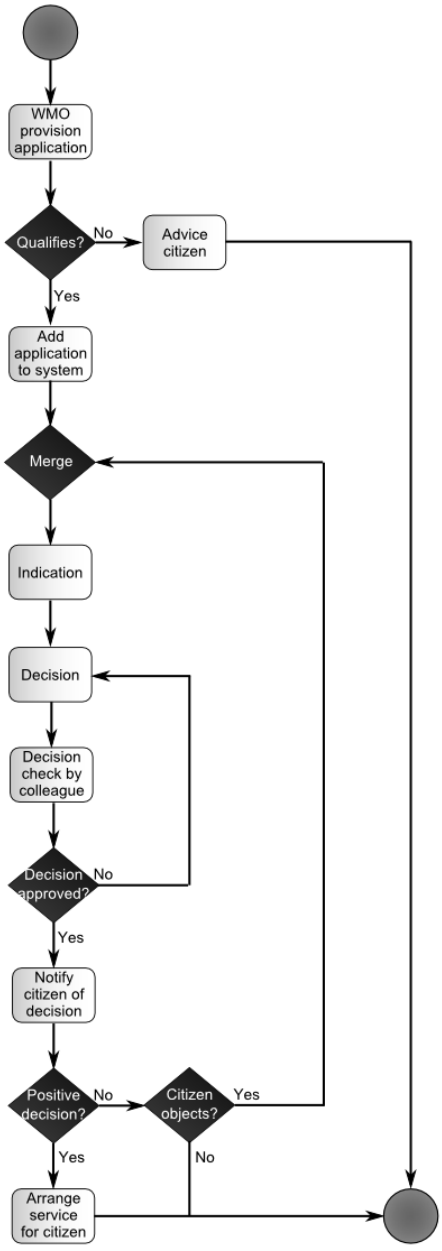


Figure 1.1: *A fragment of WMO process.*

1.1 Case–Study: Variability in Local eGovernment

The Netherlands consists of 418 municipalities which all differ greatly. Because of this, each municipality is allowed to operate independently according to their local requirements. However, all the municipalities have to provide the same services and execute the same laws. An example of such a law which is greatly subject to local needs is the WMO (Wet maatschappelijke ondersteuning, Social Support Act, 2006), a law providing needy citizens with support ranging from wheelchairs, help at home, home improvement, and shelter for the homeless.

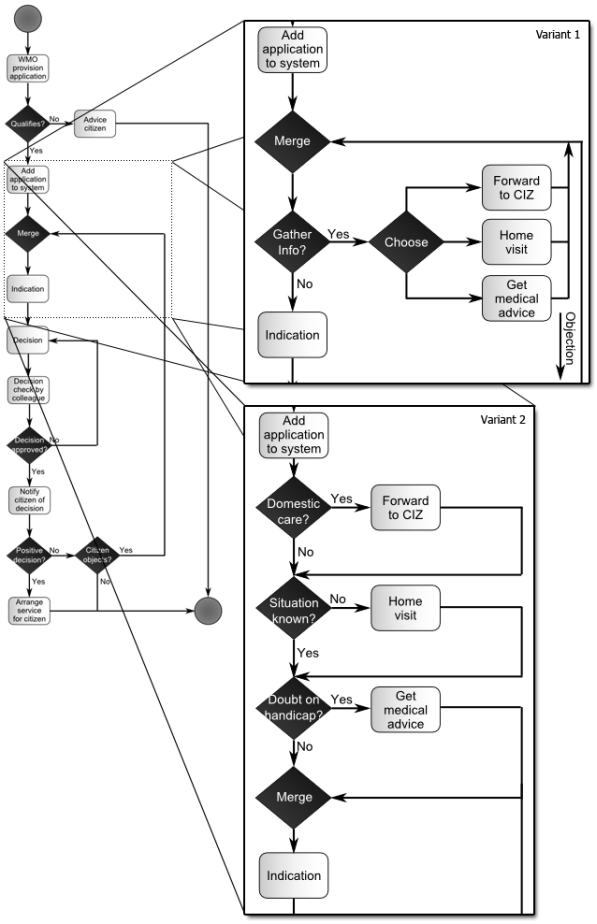


Figure 1.2: *Variability options within the WMO process.*

Figure 1.1 illustrates a simplified version of the general WMO process. The process shown here was based upon the commonalities between processes obtained

through interviews with seven different municipalities located in the Northern region of the Netherlands [van Beest, Bulanov, Wortmann and Lazovik 2010, van Beest, Kaldeli, Bulanov, Wortmann and Lazovik 2012]. Municipalities interviewed ranged in size, population, and income, and as well as differing in terms of being either urban or rural areas. The process starts with an application procedure which determines whether the request made by a citizen falls under the WMO law. If this is not the case, the citizen must be advised by the municipality employee as to what steps to take next. When the request made by the citizen does fall under the WMO law, the application is added to the system, an indication is written up, and a decision as to whether to approve requested and in what form, whether partially or in full, is made based upon the indication. The decision is then checked by a colleague, and reported back to the citizen. After this, the support is either arranged for the citizen if the decision is positive, or the citizen may object should the decision be negative.

1.1.1 Business Process Variability

Within the general WMO process as illustrated in Figure 1.1 we discovered a large amount of variability through the careful examination of the differences in the WMO processes found at the seven municipalities [van Beest, Bulanov, Wortmann and Lazovik 2010, van Beest et al. 2012]. Most notable was the variability concerning the information gathering activities illustrated in Figure 1.2, which ranged from all three the activities being included to only a home visit being included, from being in serial order to being in a loop, etc. Other variabilities included background checks on citizens, checks in the objection loop, different application mechanisms for different requests, and a decision check by a supervisor. Figure 1.2 illustrates a traditional imperative variability view concerning the information gathering activities of the WMO process. Here two variants are shown which may or may not be used within the general template. Variant 1 contains the information gathering activities in a loop, whereas Variant 2 contains them in a sequence. However, not all three information gathering activities are required to be included or, in the case of the serial variant, are required to be traversed in the same order, leading to a large number of sub–variants and increasing complexity.

1.1.2 Business Process Evolution

Another relevant problem for e–Government is the problem of business process evolution. The source of the problem lies in the fact that all of the governmental processes, despite being flexible and customizable, have to adhere to the rules imposed

by the laws. If the laws change, then all of the variants of a particular business process have to be revised and possibly modified in order to conform to the new regulations. If the requirements of the laws are implemented in the form of a business process, then all the specific implementations within local municipalities can be treated as variations of such a global business process.

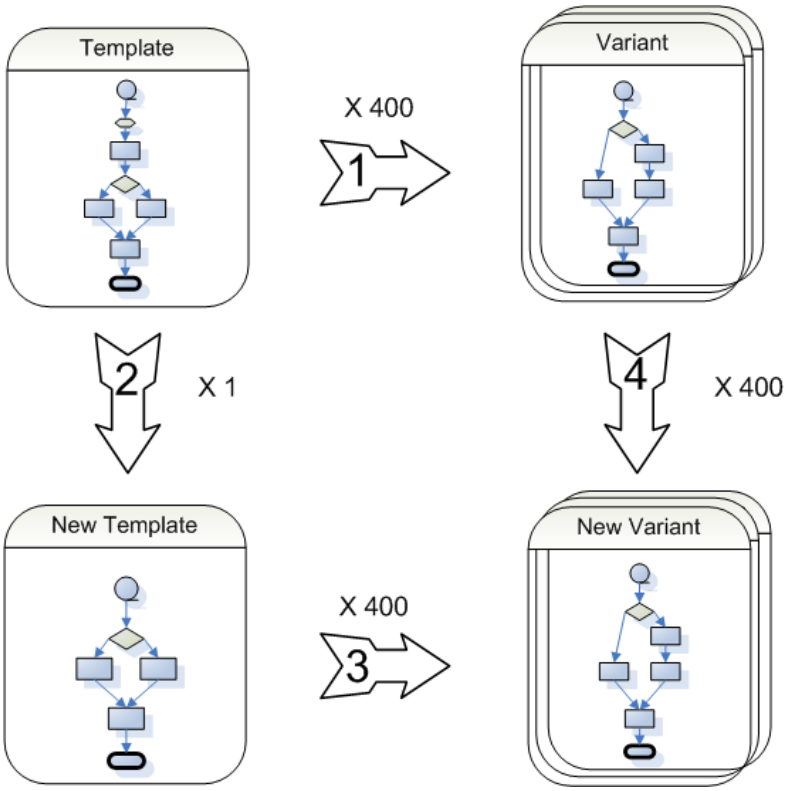


Figure 1.3: *Business Process Evolution*

This idea is illustrated in Figure 1.3, with a template process representing the global requirements in the upper left corner, and a customized one representing a variant of some local municipality in the upper right corner. The arrow "1" represents the customizations of the global template process. The number "400" indicates that such customizations can be made many times, for example, up to 400 times, since there are more that 400 municipalities in the Netherlands. In this kind of situation there is a potential danger of simultaneous modifications, when a template is modified after it has been customized (for example, in order to fix an error in the specification). This kind of modification is represented by the arrow "2" in

the figure. The problematic case in this situation arises when the customized processes (there may be many of them) must be changed in order to take into account the template modification. The arrows "3" and "4" represent two possible ways to propagate the original modifications, but in both cases that propagation must be repeated as many times as there are variants.

1.1.3 Runtime reconfiguration

Another important issue in the field of business process management is the handling of possible variations happening at run–time. Those variations are mostly occasional by their very nature, since the situations which happen frequently can be handled in the business process description.

In the scope of the WMO process, the part which is specifically prone to such runtime variations is the second part of the process, which is responsible for realization of the decisions made at the earlier stages. This assumes the involvement of third–party agents, such as suppliers and building companies.

The WMO process shown in Figure 1.4 (a fragment of this process was already presented in Figure 1.1) starts with the submission of an application for a provision by a citizen. The first part of the process concerns the procedure involved in making a decision on whether the original application is eligible or not. In case of a negative decision (i.e. the application is rejected or the granted provision is less than the citizen requested), the citizen has the option of lodging an appeal. In case of a legitimate appeal, the provision is either granted, or the process is restarted the appeal. This part of the process has already been discussed in Section 1.1.1, but, in terms of runtime reconfiguration, the second part of the process is more interesting.

The second part of WMO process starts once a decision has been made. When a decision is positive, the appropriate activities are executed, depending on the provision requested. For domestic help, the citizen has the choice between "Personal Budget" and "Care in Kind". With the choice of a "Personal Budget," the citizen periodically receives a certain amount of money for the provision granted in order to pay for workers or supervisors, and decides where the money is spent. With the choice of "Care In Kind," suppliers who can take care of the provision are contacted. A home modification involves a tender procedure to select a supplier, prior to execution of the actual home modification. A wheelchair is usually provided using a contracted supplier. After acquiring the detailed requirements, the order is sent to the supplier selected, who delivers the provision. From that point on, the process is identical for all the provisions. The order is sent to the selected supplier, who delivers the provision and sends an invoice to the municipality. Finally, the invoice is checked and paid.

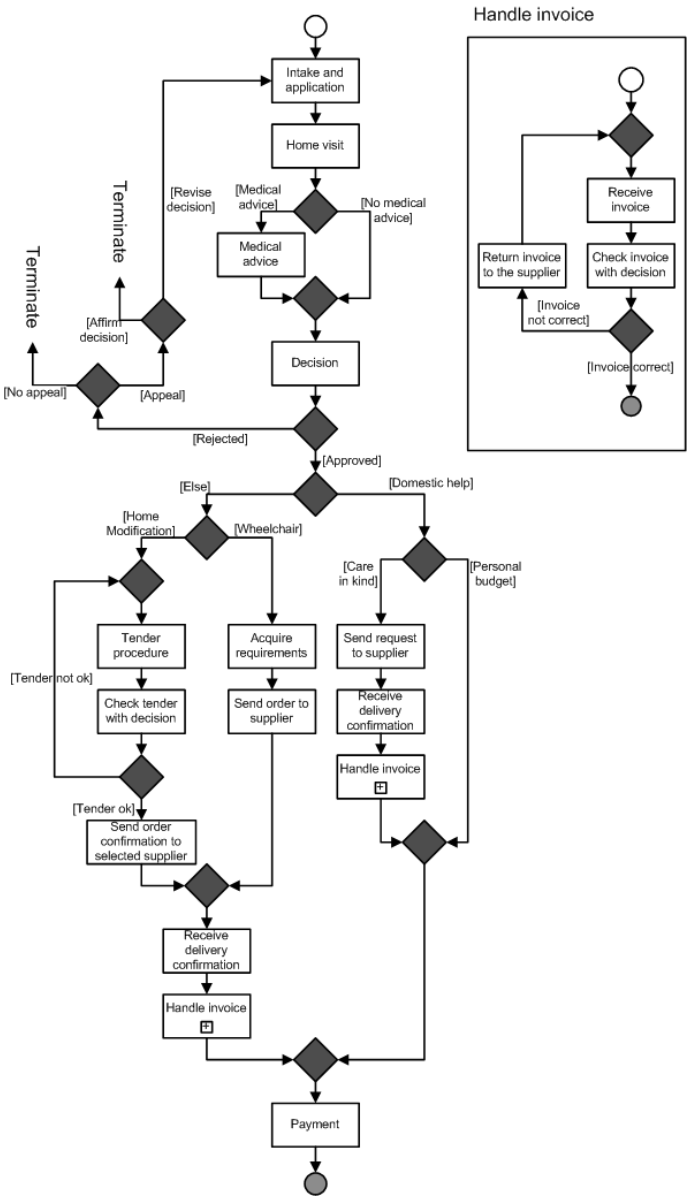


Figure 1.4: *WMO process model (BPMN notation is used).*

The request for a wheelchair or a home modification may take up to 6 weeks until the delivery of the provision. Evidently, this process implicitly depends on other stakeholders and their processes as well: The processes executed by the suppliers

of the provisions or the citizens requesting the provision can affect the process as executed by the municipalities, due to their mutual dependence on certain process variables (e.g. the address or provision specifications). The WMO process depends on the correctness of some process variables as well. However, these process variables may be changed by another process running in parallel, independent of the WMO process, and are, therefore, volatile. Regardless of whether the WMO process is designed to be proprietary to the municipality, a change in either of these volatile process variables is entirely beyond the scope of control of the municipality and may potentially have negative consequences for the WMO process. In other words, due to its dependence on those variables, these changes may result in undesirable business outcomes.

1.2 Management of Business Process Variants

The three examples discussed above are all derived from the same case study, namely, the WMO process of Dutch e-Government. Although they call for different aspects of business process management, the bottom line is nevertheless the same: it often happens in practice that a predefined process has to be modified, and such a modification can be needed only once or it may need to by applied many times over a long period.

Moreover, the processes which appear as a result of such modifications are not separate artifacts but, rather, form a family or a hierarchy. Any attempt to support such a hierarchy faces the problem of how to maintain the links between the generic processes at the top of the hierarchy and their locally customized variants.

In general, there are two approaches to deal with business process variability: imperative and declarative ones [Schonenberg, Mans, Russell, Mulyar and van der Aalst 2008]. The former approach presumes the specification of the variations which are admissible, whereas the latter one means that the boundaries on possible modifications are not specified explicitly but rather outlined by the means of additional *rules* or *constraints*. Later in the thesis we will show how to construct a solution which embraces the features of both aforementioned approaches, thus providing more possibilities for the end user.

The main contribution of this thesis is in the analysis of how to apply the formalisms of propositional and temporal logic to describe a business process model as a set of logical formulas, and in the introduction of a framework for business process variability management as a result of that analysis. The advantages of such business process representation are the following:

1. Ability to describe a whole family of business processes via correction or loosening of the formulas describing a business process. The idea is to treat any given business process model also as a **model** for temporal logic (see Appendix A), and also to represent a business process as a a set of temporal logic formulas. The representation is made in such a way that the original process model becomes the only one which satisfies all of those formulas. Any modification of the underlying business process model would make one or more of those formulas invalid.

The main advantage of our novel way to encode a business process is the ability to make simple *modifications* of the set of formulas which in turn lead to a loosening of the initial constraints. As a result, instead of a single business process model, there is a whole family of business process models which satisfy all of the formulas, and which thus allow one to make some (but not just any kind of) modifications to the original business process model. We will provide a set of simple modifications, such as making an activity optional, allowing an activity to be moved within a model, and making a *placeholder* which can be later substituted using one of the list of predefined activities. Each of those modifications has a formal definition on how to correct the initial formulas in order to achieve the desired effect.

The possibilities of those modifications are comparable with the ones provided by business process variability frameworks, such as Configurable Workflow Models [Gottschalk, van der Aalst, Jansen-Vullers and La Rosa 2008] or VxBPEL framework [Sun and Aiello 2008]. On the other hand, due to the nature of formula–based business process specification, additional constraints (which are often called *rules*, e.g., [Müller, Greiner and Rahm 2004]) can be added on the top of the original model. Such constraints describe the relationships between different activities in a business process model, such as "Activities A and B are mutually exclusive" or "Activity A must be executed after activity B." This approach has some affinity with the one presented in DecSerFlow tool [van der Aalst and Pesic 2006]; however, the prominent added value of our approach is that we naturally link such constraints with the traditional approach of utilizing a customizable reference model.

In Chapter 3, we will describe our framework for managing reference business process models, which is based on the representation of a business process as a set of formulas, and benefits from both traditional and novel constraint–based approaches to business process variability.

2. Ability to analyze the difference between two or more business process models and describe that difference in the form of transformational function. Since each of the business processes is represented as a set of formulas, we need to analyze the difference between those sets and abstract from the concrete business process models. We will utilize a business process representation in matrix form, where each row and column represents a single activity, and any cell represents the

temporal relationship between the activities which represent the corresponding row and column.

This way of representing a business process makes it possible to analyze the difference between any two given business processes as a result of matrix transformation. We will analyze different types of manipulations with matrices and how they affect the underlying business process model. In addition, we will analyze the properties of the matrices which represent business processes, and will thus show the correspondence between manipulation with matrices and the changes in business process model inflicted by such manipulation. The principle of business process representation in matrix form has some affinity with *process graphs* [La Rosa, Dumas, Uba and Dijkman 2010] which are used as an intermediate structure for merging of business process model. However, since we are aiming for a different target, we will extract the difference between business process models instead of making their composition.

Also, instead of existing temporal logics, we will utilize a novel extension which is capable of resolving the difference between the notions of "always followed by" and "sometimes followed by." Such discerning power is achieved thanks to the use of different types of branching points in a business process model, which allows one to take into account not only the analysis on the paths in a graph (as does CTL logic), but also the branching points which reside on a path together with their types (AND– split or OR–split). With the help of such logic (which we call TPL, Temporal Process Logic), we have a representation which is more descriptive than the one which is used in the field of workflow mining [van der Aalst, van Dongen, Herbst, Maruster, Schimm and Weijters 2003] and temporal planning [Ghallab, Nau and Traverso 2004].

In Chapter 4, we will formally introduce Temporal Process Logic, and then will show how this logic can be utilized to identify the difference between two business process models.

3. Ability to execute a business process so that the actual sequence of actions is decided at runtime on the basis of the context of execution. Examples of relevant contextual data are the home address of the person who initiated the process or the prices of a supplier who is chosen to fulfill the original customer's request.

In some cases, such data is crucial for the business process, since, for example, there is a difference if a request is funded by the government or not, depending on the prices of the supplier. In the business process model this is reflected as a branching point below the label "Domestic Help" in Figure 1.4. However, the decision at that point is based on the data which is provided by the supplier, and such data may change over time.

Imagine the situation where the execution has followed the left branch and the

activity "Send request to supplier" is in progress; and at that moment the situation has changed, and the right branch should have been taken instead, according to the new information. In practice, however, the execution of the business process will continue following the wrong path, and the mistake will be discovered after the process has finished.

This situation can be remedied if the business process execution engine has the following features: (i) it listens to the data modification events, and (ii), if necessary, stops the execution of the business process and starts a so–called *repair process*. Such a repair process could contain the activities of the alternative branch of the business process together with the activities which compensate for the result of the activities which were already executed. Alternatively, such a repair process might not even resemble the original process and might be used in very specific situations only.

Additionally, since a business process is specified as a set of formulas, they can be used as input for the generation of a business process automatically, using automated planning techniques [Ghallab et al. 2004]. The additional feature of using planning is the ability to generate a business process which is different from the original one because of the changes in the original formulas. Such changes in the formulas are driven by the modifications of the data a business process has relied on.

In Chapter 5, we will describe the architecture of a business process execution engine which possesses both of the features (i) and (ii) just discussed in the previous paragraph, and which uses automated planning techniques in order to generate a repair process. Also, we will show how to encode a business process and its execution context into a structure which is understandable by a planner, and conversely, how to execute the outcome of the planner as a part of the business process.

The rest of the thesis is organized as follows. In Chapter 2, we will give an overview of the state of the art research in the field of business process variability and reconfiguration. In Chapter 3, we will introduce our framework for designing business process templates with explicit support for variability. In Chapter 4, we will introduce a new kind of modal logic which is capable of expressing the temporal relationships in a business process model, taking into account different types of branching points. This logic will be used to analyze how to establish a link between a generic process and its variants, which thus provides the options to maintain the evolution of business processes. In Chapter 5 we will describe how to deal with runtime business process reconfiguration with the help of automated planning techniques [Ghallab et al. 2004]. Finally, in Chapter 6, we will provide a brief summary of this thesis and outline a few directions for future research.

Partly published as:
M. Aiello and P. Bulanov and H. Groefsema – *"Requirements and Tools for Variability Management,"* IEEE workshop on Requirement Engineering for Services (REFS 2010) at IEEE COMPSAC, pp. 245–250, 2010.

Chapter 2

Related Work

In software engineering, *variability* refers to the possibility of changes in software products and models [Sinnema et al. 2006], and the concept of variability is considered to be important in the field of software product lines engineering [Pohl et al. 2005, Clements 2006, Galvão, van den Broek and Akşit 2010].

In the context of BPM, variability indicates that parts of a business process remain variable or not fully defined in order to support different versions of the same process, depending on the intended use or execution context. This kind of variability is often included through the introduction of so-called *variation points*, that is, elements of a business process where change may occur. A process in which variability is included is called a *reference* or *generic process*. Processes where choices have been made that are derived from the reference process are called *variants*. The strategies for arriving at a reference process can differ greatly and range from drafting a simple template based on commonalities between variants to using a single variant as a reference process. *Variability management* is the set of activities designed to cover the creation and support of differences in versions of reference processes.

The advantage of explicit variability treatment for modeling and managing business processes resides in the reuse of the reference processes and in the handling of evolutions and customizations. Instead of creating entirely different processes for each variant and thus introducing redundancy and the possibility of errors, variability offers the introduction of these variants through explicit traversal choices, thus eliminating duplication of work and a source of inconsistencies across versions. Another paramount advantage lies in the readability and maintainability of the processes. Instead of using BP constructs within a process to model the variations in circumstances explicitly, and thus introducing readability and maintainability problems, variability offers a clean solution through the introduction of variants.

Finally, we should note that *variability* is very closely related to *flexibility*. Flexibility offers adaptation and change for a process, whereas variability deals with different versions of a process. Of course, in order to support different versions, a certain amount of flexibility and change management is required. Through the

advantages that flexibility offers in BPM, the support for different versions of the process can be offered by the variability tools. In other words, flexibility is the tool that allows the different versions of the process to be created and managed.

2.1 Business Process Variability

Variability management is an extension of the typical activities involved in business process management. We give a general depiction in Figure 2.1. On the left, one can see how requirements drive the definition of the design processes. Once the designer has a first design of the process, then they move on to the deployment and run-time phases. In case of errors, unpredictable situations and changes in requirements, the whole procedure must be repeated, starting from the design stage. During execution, information on normal and exceptional execution is collected (Monitoring and Diagnostic phases), thus allowing for more flexible process support and feedback for process evolution. Variability management complements these general BPM phases by introducing a set of parallel stages, as seen on the right in the figure. First, a decision is made as to the types of variability and at which stage of the lifecycle they should be introduced. In this context, there are two main categories: design-time and run-time variability.

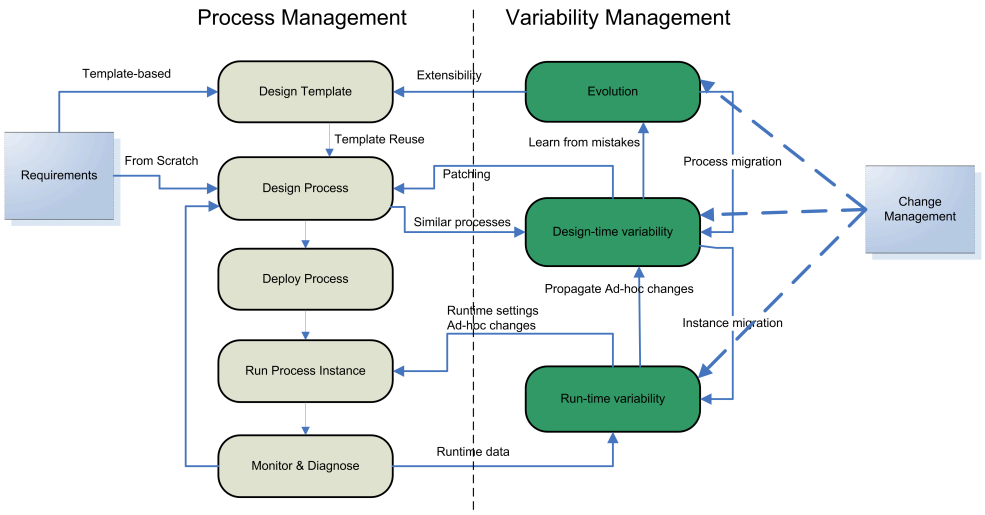


Figure 2.1: *Process lifecycle and variability management.*

Design–time variability deals with the definition of variations in processes at design–time. Similar processes are implemented using a reference process and applying

different variations in order to support all variants. Often this means finding the commonalities between the similar processes being implemented and introducing variations where differences occur. In addition, it may mean taking a process and foreseeing all possible customization and changes that different contexts may require. The next step is designing the variations for the generic process in such a way that they cover all variants identified in the requirements through the flexibility offered by the variability. New variants are added by reusing of the existing reference process or patching existing variants. The source of such changes might be either changes in requirements or propagation of a change at a different level.

Run–time variability is responsible for managing variation of processes in execution. The major issue it addresses is handling redesigns of running processes. Such redesigns can range from skipping/deleting a single activity to moving to a whole different variant. The source of changes might be either changes in requirements, responding to an erroneous situation (via analyzing run–time data), or a propagation of a change at a higher variability level (design–time or evolution).

A cross-cutting element of variability management is the evolution that generic processes go through. *Evolution variability management*, in black on the right of Figure 2.1, represents the changes introduced not by customization but rather by changes occurring over time.

2.1.1 Declarative vs. imperative variability

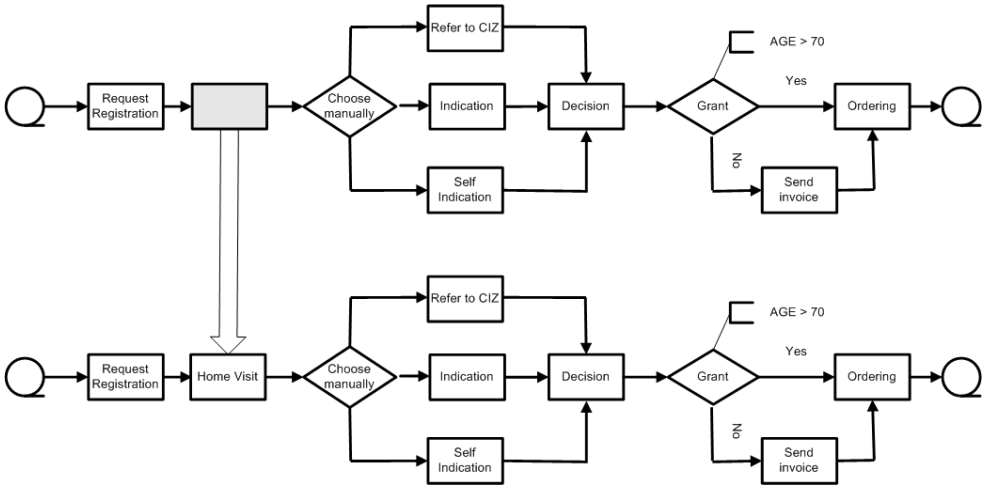


Figure 2.2: *An imperative template process.*

When arriving at a particular technique to address the problem of variability,

the task of specification of the template process and its admissible modifications arises. In general, the way to address this problem is a core part of any BP variability platform. Despite the fact that the solutions may vary greatly, a general taxonomy may still be introduced. In [Schonenberg et al. 2008], the authors introduce two major categories: imperative and declarative ones. In the following sections those categories will be described and compared.

Imperative

The traditional way to describe a business process arises from imperative programming languages, where the source code of a program is a formal computer–readable representation of some predefined sequence of actions. Similarly, modern BP modeling languages like BPMN are basically visual programming languages, suitable for sketching a visual schematic representation of a given algorithm or a workflow.

Concerning the problem of change management, the solution is also inherited from the area of software engineering [Sinnema et al. 2006, Bachmann, Goedicke, Leite, Nord, Pohl, Ramesh and Vilbig 2004], which means that possible process modifications have to be anticipated in advance and specified in the form of *variability points*. Those variability points are used later in order to customize the template and obtain a customized process which is often called a *variant*. Typically imperative–oriented variability can be represented as a template process with several variability points specified in it. Figure 2.2 shows an example of an imperative–based template, with the variability point in the form of a placeholder.

This placeholder indicates the place in the business process model which is left unspecified. As a result, it becomes possible to create several variants based on the same template by choosing an appropriate activity to be put in place of the placeholder. In Figure 2.2, the process at the bottom illustrates one of the possible variants, which is made by selecting the activity “Home Visit” to fill the placeholder. However, when using this approach, a highly flexible business process either contains a lot of variability points over the same template or requires a combination with other variability techniques.

Nevertheless, this approach has several important advantages. First of all, it is intuitively close to the well–known standards of business process modeling, such as BPMN [Object Management Group (OMG) 2009]. Second, it is usually easy to implement on top of existing business process engines. And, finally, the validity of customized processes can be guaranteed as long as the template is verified. The verification of a template in that case means the testing of all the combinations of all variability points in order to check whether the corresponding process variant is acceptable or not. However, the need to specify variation points implies unneces-

sary restrictions imposed on the business processes [van der Aalst, ter Hofstede and Weske 2003].

Declarative

The main idea of the declarative approach to business process variability is to specify the boundaries which confine the possible modifications instead of focusing on the process model itself. Such boundaries are usually represented as a set of formulas of propositional or modal temporal logic which are typically called *rules* or *constraints*. The validity of a particular process variant means that all of the constraints are valid for that variant. As a result, a set of possible valid variants is implicitly defined by a set of constraints, with no need to specify them one by one.

In a declarative process management framework, a generic process usually consists of a reference business process and a set of constraints on top of it [Momotko and Subieta 2004, Charfi and Mezini 2004], thus providing a starting point for process customizations. In some cases, however, the generic process is not specified at all, and a variant must be built on the fly by a process designer or even by an end–user during the actual process execution [Pesic, Schonenberg, Sidorova and van der Aalst 2007, Lu, Sadiq and Governatori 2009]. In the latter case, the precise structure of a business process remains undefined until the very last moment, which can be considered being a guided process execution.

The constraints can be roughly divided into two categories: *selection* and *execution* ones. Constraints of the first category prescribe which of the activities must or must not be selected to be the part of a particular process variant. Such constraints are usually expressed as formulas of propositional logic, which makes it possbile not only to specify which activities to select but also to define their interdependencies [Sun and Aiello 2008].

The second category is used to specify the execution order (at run–time). Constraints of this category utilize temporal logic, typically LTL or CTL[1]. The former type of formulas allow the order of the execution to be bounded, for example, to disallow the execution of the activity “X” unless the activity “X1” has been executed. This is utilized, for instance, in DECLARE tool [Pesic et al. 2007]. The latter type of formulas deals with branching time flow, which means taking into account alternative execution paths. In practice, that means the ability to control the design–time customization of business process models, where it is important to distinguish between the paths which will always be followed at run–time and the paths which are optional but still can be taken.

[1]See appendix A

Discussion/Comparison

Comparing both of the previously discussed approaches, the declarative one in general provides more flexibility for business process designers, since some parts of the process may be left either undecided or with minor restrictions only, thus leaving the final decision to the end user. At the same time, other parts of the same process may be covered by constraints, thus leaving only small margins for possible modifications.

One of the disadvantages of the declarative approach lies in the fact that constraints may contradict each other or restrict some options which would be otherwise acceptable. To solve this issue, the set of formulas can be formally verified in order to identify whether there is at least one business process model which satisfies all of the formulas. More complicated solutions involve the generation of possible valid business process models. In any case, the underlying task is the problem of boolean satisfiability, which is known to be NP–complete [Cook 1971]. Therefore, such verifications are problematic in practice unless some restrictions are introduced in order to simplify the problem [Dijkman, La Rosa and Reijers 2012].

Another disadvantage of declarative approaches lies in the semantic gap between a business process model and declarative constraints. The logical formulas are introduced by means of various declarative techniques and must be attached in some way to the business process which is under consideration. But the problem is that business processes are typically specified using one of the traditional imperative business process notations which typically provide no means to specify any additional constraints. The task of connecting such logical formulas with the process model is therefore transferred to business process designers.

There are, however, several attempts to overcome this issue by virtue of the introduction of additional visual elements along with the translation of those elements into low–level formulas. These elements may range from simple arrows to complicated grouping elements embracing whole areas in a business process model. For example, an *ordering* constraint can be represented visually as an arrow connecting the two activities it embraces, and a *selection* constraint can be represented as an additional property or as a tick box [Gottschalk et al. 2008, Pesic et al. 2007, van der Aalst and Pesic 2006, Lu et al. 2009, Groefsema, Bulanov and Aiello 2011].

2.1.2 Annotation–based Variability

In many cases business processes inside of an organization are well established and driven by laws or strict policies, and therefore only minor and predictable modifications are allowed. In such cases it becomes feasible to describe admissible variations such as these and store them along with the original business process model. This

provides total control over the possible modifications, and also makes it possible to perform an automated business process customization.

Configurable Workflow Models

An example of such a variability is presented in [Gottschalk et al. 2008] in the form of configurable workflow models. Such models involve a generic business process model together with a set of formally defined *requirements* and *guidelines* which can be used in order to generate a particular business process variant. In Figure 2.3, in the top section, an example of such a configurable model is represented. Here, along with the generic business process model, there is one *guideline* and two *requirements*. The guideline specifies that the activity "Home Visit" must not be included if the population of the city exceeds 5000 people (this information is presumed to be known as a part of general domain knowledge). One of the two requirements states that the absence of the "Home Visit" activity means that the "Self Indication" activity must not be in the process. The second requirement states that the activities "Self Indication" and "Indication" are mutually exclusive and cannot both be included in the same business process. The second requirement also means that the generic process itself is not valid and therefore cannot be used as is without prior customization.

In Figure 2.3, bottom section, one of the possible variants is displayed. Here the activity "Home Visit" is omitted, possibly because of the size of the city. This choice implies the exclusion of the "Self Indication" activity; the activity "Refer to CIZ" is not touched, since there are no guidelines or requirements concerning this activity.

A similar idea of utilization of annotation–based variability is explored in [Becker, Delfmann and Knackstedt 2007, Delfmann and Knackstedt 2007]. The authors introduce the idea of *model projection*, where a template model is enhanced with a set of annotations, and a concrete business process variant is derived via so–called model projection step. At this step, a transformation function decides which pieces of the original template must or must not be included in the final variant basing on the annotations in the template process.

A further exploration of annotation–based variability lies in the ability to build a business process model on the fly using a query language. In [Momotko and Subieta 2004], the authors introduce BPQL language (Business Process Query Language), which allows a user to build a query in order to specify the characteristics of the desired business process variant, and the system will build one besed on the annotations in business process templates in a process repository.

Another extension to this approach is to attach the requirements and guidelines to the list of the *features* [Kang, Cohen, Hess, Novak and Peterson 1990, Kang, Lee

and Donohoe 2002, Czarnecki and Eisenecker 2000] of the system. An end–user may therefore select a set of desired features instead of working with low–level requirements attached to the template business process model [Lapouchnian, Yu and Mylopoulos 2007, La Rosa, van der Aalst, Dumas and ter Hofstede 2009, La Rosa, Lux, Seidel, Dumas and ter Hofstede 2007, Schnieders and Puhlmann 2007].

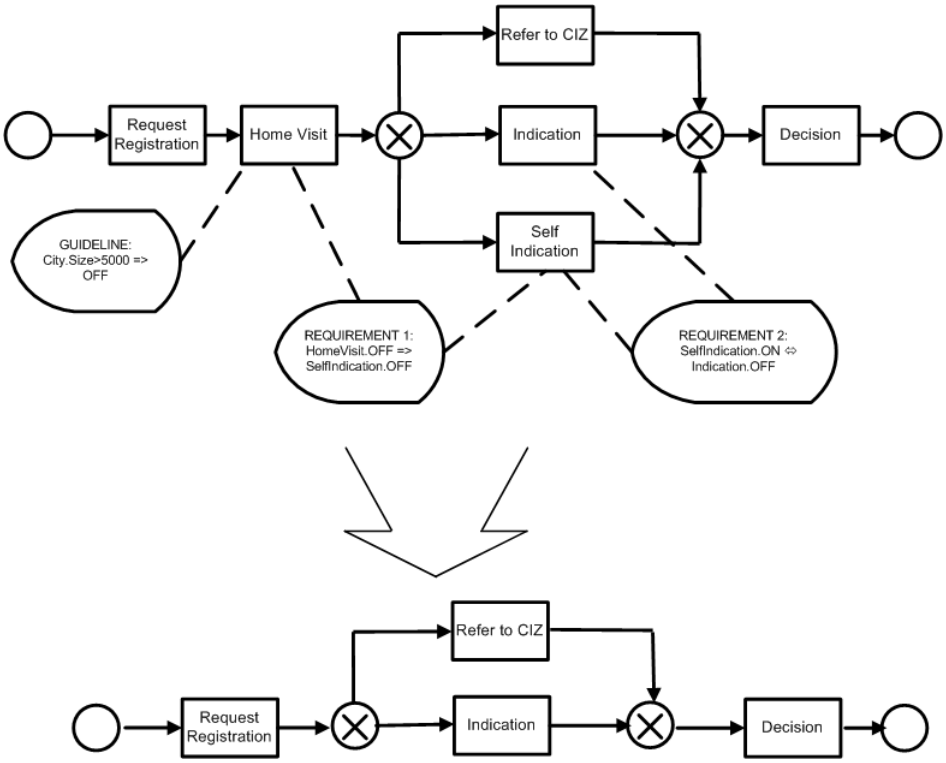


Figure 2.3: *A configurable workflow model example.*

Flexibility based on Variability Points

An example of traditional imperative approach based on variability points is VxBPEL [Sun and Aiello 2008]. A single variability point is an area in a business process model plus a list of options available for that variability point. For example, a variability point may embrace a single activity, and the list of options in that case would be a list of substitute activities, possibly including an "empty one" (which would mean the removal of the underlying activity).

In Figure 2.4 a business process model with two variability points is illustrated.

The first one (VP1) encloses a single placeholder and offers three possible options to be chosen (shown in the lower–left part of the figure). Of those three options, options 1 and 3 represent single activities, while Option 2 contains a whole sub–process built of two sequential activities. The second variability point (VP2) is responsible for the particular choice condition in the only choice point in the process (which is usually called gateway). Of the two possible options, Option 1 is also declared to be mutually exclusive with the Option 3 of the VP1, which is illustrated as a two–way arrow with a cross in the middle.

Also, a notable example of a framework based on explicit variability points is PROVOP [Hallerbach, Bauer and Reichert 2008]. A business process in PROVOP is enhanced with a list of **options**, and each option is associated with a list of necessary **process changes**. Each of such changes is in turn linked with a particular place in the business process model (such places are called *adjustment points*). The four types of changes supported by PROVOP are: **insert** a fragment, **remove** a fragment, **move** a fragment, and **modify** an attribute. In result, in order to obtain a concrete business process variant one needs to chose one or more of the **options** to be applied, and the system will perform the process changes specified behind those chosen options.

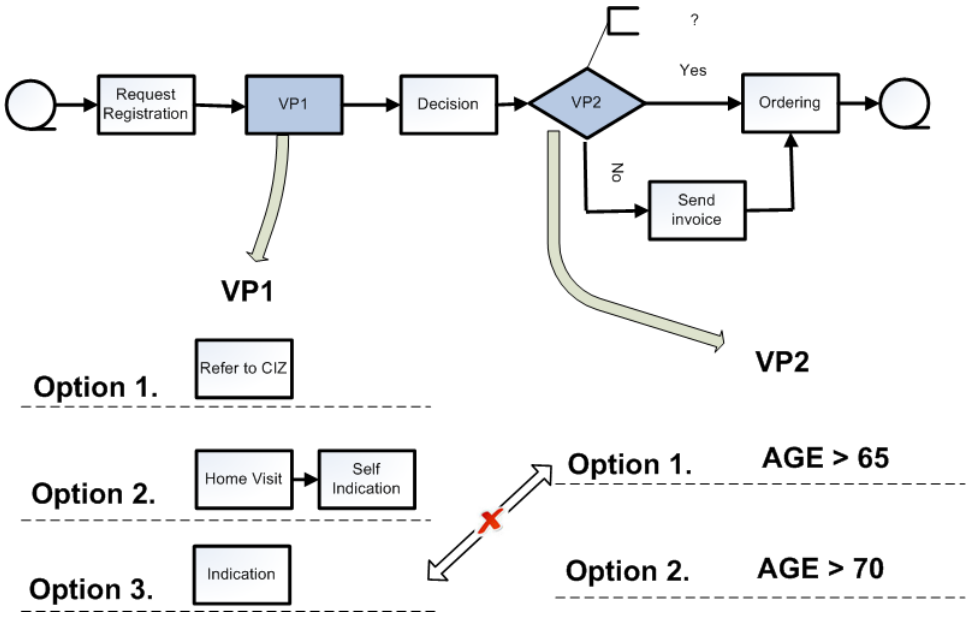


Figure 2.4: *Variability points using VxBPEL.*

2.1.3 Variability by Underspecification

In the case of business process modeling, *underspecification* means that some parts of a process model are left undefined. According to [Weber, Reichert and Rinderle-Ma 2008], there are three major classes of variability by underspecification: **late binding**, **late modeling**, and **late composition**. A simple example of the first class is illustrated in Figure 2.2, where the grey unnamed rectangle indicates the place where some activity might be inserted. Once all placeholders are filled, the appropriate business process model is ready to be consumed. The concept of late binding has been utilized in many business process management system, including *Worklets* [Adams, ter Hofstede, Edmond and van der Aalst 2006] and *MOBILE* [Jablonski 1994].

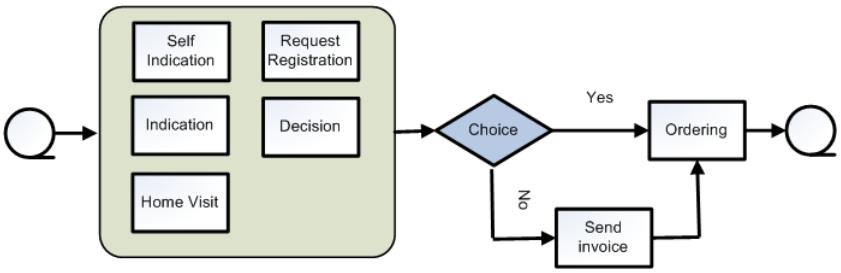


Figure 2.5: *A business process with a pocket of variability in it.*

Business Process Constraint Framework (BPCN, [Lu et al. 2009]) is an example of the second class (late modeling). In Figure 2.5, an example of a not fully defined business process is presented. The grayed rectangle in the left part of the process is a **pocket of variability** [Sadiq, Orlowska and Sadiq 2005] with its content left undefined. On the contrary, the rest of the process is fixed and not supposed to be changed. The activities displayed inside of this pocket of variability are the ones which can be used to build a customized process, but the exact structure of the final process is also restricted by the means of constraints.

There are two types of constraints in BPCN: **selection** and **scheduling** ones. Constraints of the former type restrict which activities can (or must) be selected and which cannot, depending also on which activities were already chosen. Examples of selection constraints are **mandatory** (self–descriptive), **cardinality** (specifies how many occurences of a given activity are allowed), or **substitution** (if an activity "A" was not selected, then some activity "B" must be selected instead).

Constraints of the latter type restrict the execution order of the activities in the customized business process. The execution of a single activity is considered as a temporal interval, and scheduling constraints are essentially possible interrelations

between those temporal intervals [Allen 1983]. Examples of scheduling constraints are **before** or **parallel** (both are self–descriptive).

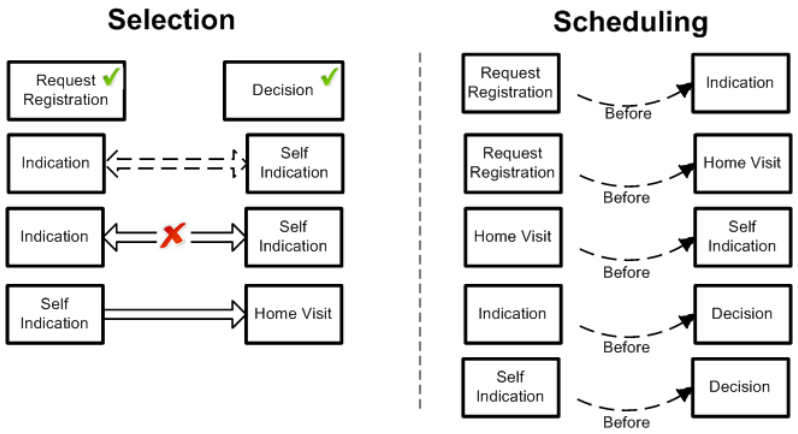


Figure 2.6: *Sets of constraints.*

An example of a possible set of constraints for the generic process of Figure 2.5 is illustrated in Figure 2.6. The constraints are divided into selection (left side) and scheduling ones. Looking at the scheduling constraints line by line, the first line contains two mandatory activities, "Request Registration" and "Decision," each bearing a tick in the upper–right corner to illustrate **mandatory** constraints. The second line illustrates a bilateral **substitution** constraint between the activities "Indication" and "Self Indication," which implies that at least one of those activities must be selected. The third line illustrates a mutual **exclusion** constraint stating that if the activity "Indication" is selected then the "Self Indication" one must not be selected, and vice versa. And finally, the fourth line illustrates an **inclusion** constraint, which states that if "Self Indication" is included, then "Home Visit" must be included as well.

But the selection constraints alone cannot describe the possible relative order of the activities in the final process. For that purpose, scheduling constraints have to be considered. In Figure 2.6, right, those constraints are represented. They are all of type "Before," which prescribes a simple ordering of the two activities it bounds. In result, the activity "Request Registration" must go before the "Indication" and "Home Visit" ones, and so on.

In Figure 2.7 three possible business process fragments are represented, and each of them satisfies all of the constraints displayed in Figure 2.6.

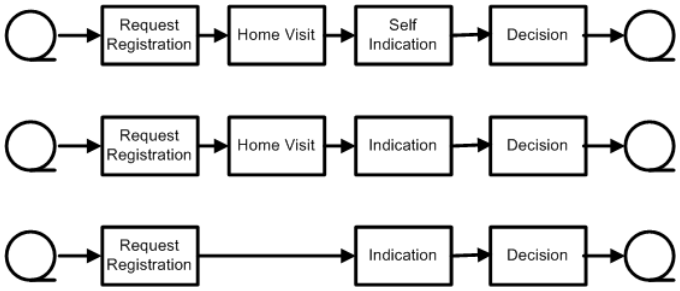


Figure 2.7: *Possible business process variants satisfying the constraints of Figure 2.6.*

2.1.4 Declarative Process Execution

When one needs to specify a business process, there is an implicit convention that one of the state of the art business process modeling languages (like BPMN [Object Management Group (OMG) 2009] or UML Activity Diagrams [(OMG) 2005]) should be used. However, in that case a process model becomes strictly defined, and any deviation from the original flow of events would be problematic, because there is no simple way to specify all the possible deviations in advance using the languages mentioned above. A declarative approach to business process modeling offers more flexibility, but on the other hand the need to specify the rules in the formal way poses an additional burden to the people responsible for business process modeling.

The DECLARE tool offers a way to make a declarative–based business process model visually [Pesic et al. 2007]. This is achieved via an open set of constraints, with a visual interpretation for each of them. Each constraint is a formula of linear temporal logic (LTL[2]). For each activity in a process a dedicated propositional variable is introduced, and for simplicity the name of this variable is the same as the name of its corresponding activity. Such a simplification is unambiguous, since there is a one–to–one mapping between the set of activities and the set of propositional variables.

Internally a business process is represented in the form of an automaton [Giannakopoulou and Havelund 2001], where each transition corresponds to the execution of a particular activity, and each state represents the state of the process between the executions, including the initial state. At each state, all the variables are equal to FALSE, apart from the variable which corresponds to the activity which has just been executed. In other words, the activity whose execution led to the current state has its corresponding variable equal to TRUE, and all other variables are equal to FALSE. It follows from this definition that all the variables are equal to FALSE at the initial state.

[2]See Appendix A

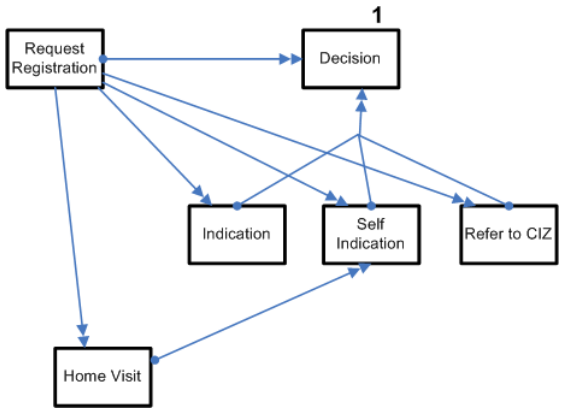


Figure 2.8: *A flexible business process definition using DECLARE tool.*

Such automaton can be used as a frame for LTL (e.g., [Blackburn, Rijke and Venema 2001]), and at each state all of the constraints are evaluated in order to decide if that state is valid or not. If any of the constraints is violated, then the state is marked as **temporarily violated**. If all of the states which are reachable from the current state are **violated**, then the state is marked as **permanently violated**. A transition from one state to another is allowed only if the target state is not **permanently violated**, which makes it possible to filter the list of activities which are allowed to be executed at the current state, thus providing a kind of guided business process execution from the user's point of view. A sequence of valid transitions forms an **execution trace** of a particular business process. Generally, there may be more than one execution traces available for a given process, and none of them must be finite. It must be noted that a business process execution can be stopped at any time at a **valid** state, therefore, the final goal of any execution is to reach a **valid** state.

Name	Visual Element	Formal meaning
EXACTLY_1		$\diamond A \wedge \Box(A \Rightarrow \Box\neg A)$
PRECEDENCE		$\diamond B \Rightarrow (\neg B \cup A)$
RESPONSE		$\Box(A \Rightarrow \diamond B)$
SUCCESSION		Combination of PRECEDENCE and RESPONSE

Table 2.1: *Constraints used in Figure 2.8.*

Figure 2.8 shows an example of a constraint–based business process model. All of the constraint used to describe the business process are listed in Table 2.1. The first constraint is called EXACTLY_1, and it means that the corresponding activity must be executed exactly once. In the model, it is illustrated as a label "1" above the activity box, and in Figure 2.8 the activity "Decision" is marked with this constraint. In practice, it means that the goal of the whole process is to get the "Decision" activity executed, and it must not be executed for the second time. However, before this activity can be executed, some pre–requisites must be fulfilled, and the rest of the model describes how to attain this.

The second line in Table 2.1 describes the PRECEDENCE constraint, which for the pair of activities "A" and "B" means that the activity "A" must be executed before the activity "B," but if the activity "B" is never executed then the activity "A" is not required.

The third line describes the RESPONSE constraint, which for a pair of activities "A" and "B" means that if the activity "A" is executed, then the activity "B" must also be executed at some point (but after the activity "A"). And the final line describes the SUCCESSION constraint, which is basically the combination of the former two constraints.

In Figure 2.8, the activities "Request Registration" and "Decision" are linked with a SUCCESSION constraint, which means that the activity "Request Registration" is a pre–requisite for the activity "Decision." Since there is a constraint that the activity "Decision" must be executed exactly once, it implies that the activity "Request Registration" must also be executed at some point. Next, there are constraints of type PRECEDENCE between "Request Registration" and each of the other activities, which means it must be executed before any of the others. Next, there is a constraint of type SUCCESSION between "Home Visit" and "Self Indication," thus making those two activities "mutual friends", since there must be either both or none of them executed, and "Home Visit" must be executed first. And finally, there is a forked arrow of type SUCCESSION linking the triple of activities ("Indication," "Self Indication," "Refer to CIZ") and the activity "Decision." It means that any of the former three activities must be eventually followed by the "Decision" one, and the "Decision" must be preceded by at least one of those three.

Figure 2.9 displays one of the possible execution traces for the business process model in Figure 2.8. Since the activity "Request Registration" is constrained to be executed before any other, it is the only one candidate to be executed at the initial state. Then, for example, the "Indication" one can be executed. At that moment, the "Decision" activity must be available for execution, which would lead to a **valid** state followed by the termination of the process execution. But since this is not compulsory, a user can choose to execute the "Home Visit" instead, followed by the "Refer

to CIZ" one. But now the activity "Self Indication" becomes mandatory because of the SUCCESSION constraint between the "Home Visit" and "Self Indication." And finally, once the "Self Indication" activity has been executed, it becomes possible to execute the "Decision" one and to terminate the process.

A further extension to the idea of declarative process execution is presented in [van der Aalst and Pesic 2006], where the authors introduce the *DecSerFlow* tool, which has an extended set of constraints as well as a set of interrelations between the constraints. Those interrelations give the ability to make particular constraints mandatory or optional depending on the current state and which activities had been executed that far.

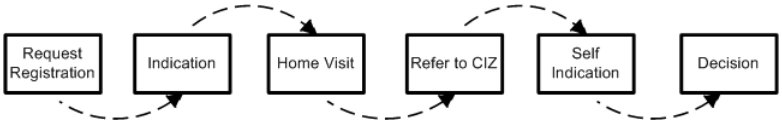


Figure 2.9: *One of the possible execution traces for the process in Figure 2.8.*

2.1.5 Runtime Business Process Reconfiguration

In addition to the task of managing possible run–time deviations, there is a task of migration of an already running business process instance to a new business process model. This task is a difficult problem per se, since the state of a running instance may be completely incompatible with the new business process model to be applied [Rinderle, Reichert and Dadam 2004]. The $ADEPT_{FLEX}$ [Reichert and Dadam 1998, Dadam and Reichert 2009] framework provides a comprehensive solution for dealing with run–time reconfiguration. This is achieved via consideration of different change patterns which may occur during the process reconfiguration, taking into account the implicit dependencies which arise from the flow of the data. Another way to address the problem of runtime reconfiguration is to specify a business process model in such a way that certain fragments of the process become available or not available depending on the execution environment and user's decisions [van der Aalst, Weske and Grünbauer 2005].

To cover for process execution inconsistencies, a number of techniques have been proposed. AGENTWORK is a workflow management system which supports automated business process adaptations in a comprehensive way, when possible exceptions and necessary workflow adaptations are specified through a rule-based approach. Using this approach, the system is able to react to process-failures like unavailable resources or data [Müller et al. 2004]. Similarly, existing runtime solutions for process interference are based on failing processes as well, e.g. [Garcia-Molina

and Salem 1987, Xiao and Urban 2008, Gajewski, Meyer, Momotko, Schuschel and Weske 2005].

A more elaborate solution for process interference in Service-Oriented Computing is provided by [Urban, Gao, Shrestha and Courter 2011]. Predefined (design-time) rules are used to specify the required compensation actions in case of interference. In addition to failing processes, this approach incorporates events like exceptional conditions or unavailable activities.

2.2 Business Process Generation

When some real–world business process is being automated, somebody has to describe that process formally and create a business process model. But in some cases such a model can be created automatically basing on some additional information, like existing business process models made for solving similar tasks, or statistical data taken from system logs. Later in this section, several approaches for automated business process generation are discussed.

2.2.1 Process Merge

One way of creating a new business process is by the combination of two existing ones which naturally should retain the behavioral features of both original processes. Examples of such process merge span from the merging of two companies which effectively leads to the merging of their workflows to the task of coordinating of multiple business process versions spread over different municipalities.

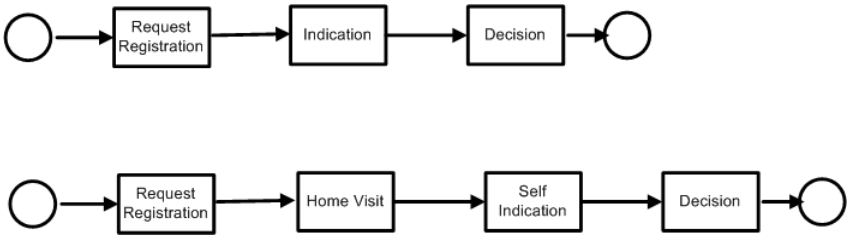


Figure 2.10: *Two examples of business processes to be merged.*

Merging using Supplementary Structures

One of the ways to make such a merge is described in [La Rosa et al. 2010], with the application for the processes represented in the form of EPC [Sarshar and Loos 2005]. The main principle is to derive a so–called **function graph** for each of the

input processes, and then make a combination of those graphs. A **function graph** is essentially a simplified representation of a business process, with one–to–one mapping between activities of the input process and nodes of the result graph. The gateways of the original process are not translated into the function graph. Instead, each arc of the graph, apart from the fact that it represents the existence of appropriate path in the original business process, also bears two labels. The labels represent the type of splitting and type of joining, respectively. The rules how to make those labels are described in [La Rosa et al. 2010], but in general they are determined by the gateways lying between the activities and their types.

In Figure 2.10, there are two business process models which need to be merged. They can represent, for example, two different implementations of the same generic business process which were taken from different municipalities, and the task is to obtain a combined business process comprising the features of both implementations. First, a **function graph** has to be built for each of the original processes. Since both of them do not contain gateways, the resulting function graphs actually repeat the structure of their original business process model and therefore are not displayed.

The next step is to make a united function graph as a combination of the original ones. The rules of the merging are the following:

1. the set of activities of the result graph is the union of the appropriate sets of the original graphs;

2. the set of arcs is the union of the appropriate sets of the original graphs;

3. the split– and join– labels are computed according to the algorithm described in [La Rosa et al. 2010].

The result of the merging is illustrated in Figure 2.11, (A), there a split–labels is attached to the beginning of each arc, and a join–label is attached to the head of each arc. The arcs with no labels actually have simple labels: both split– and join– labels are of type "AND." However, a function graph is not a proper business process, and it must be transformed into a business process model. Authors use EPC for that purpose, although BPMN notation can by utilized as well. The transformation itself is divided into two steps. First, a function graph is transformed into an EPC model via adding the gateways, and the types of those gateways are driven by the types of corresponding split–labels. If the types of split–labels of all arcs leading from some activity are the same, then a gateway of that type is added, otherwise, an OR–gateway is added instead (this is considered to be a default gateway type). The same applies for the join gateways, but join–labels are considered instead. The result of such transformation is illustrated in Figure 2.11, (B). Since the

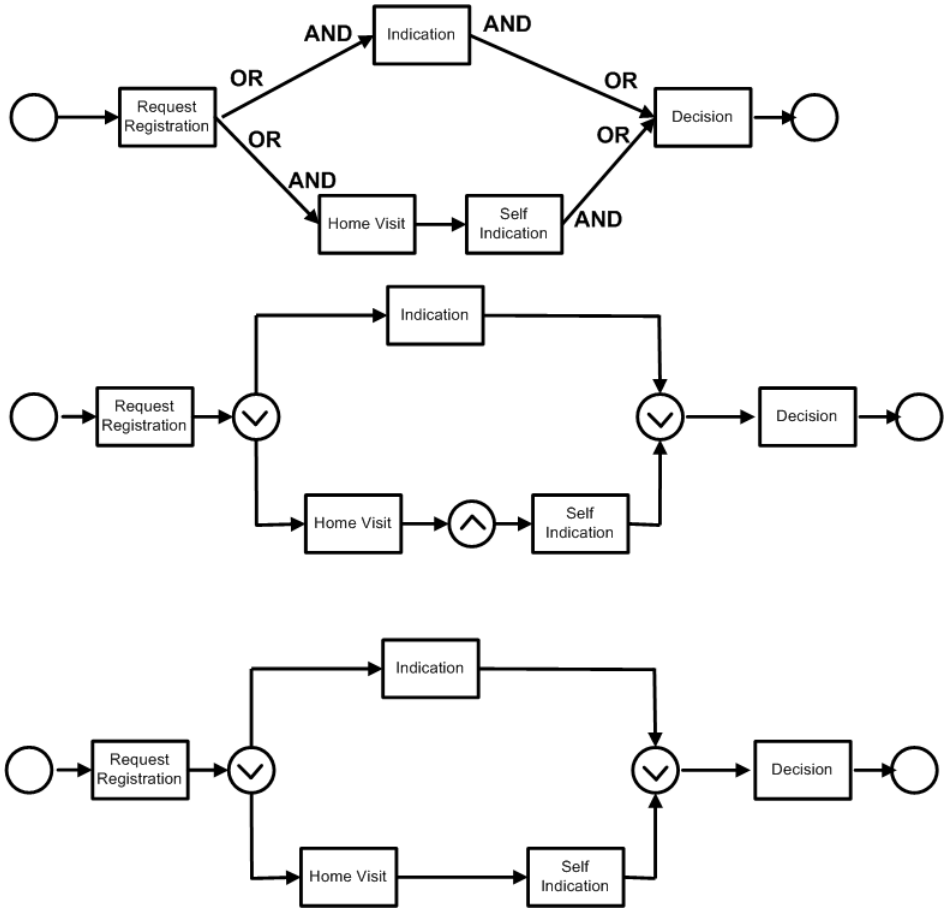


Figure 2.11: *The steps of business process merging.*

result may contain redundant gateways (and does contain, since the gateway between the "Home Visit" and "Request Indication" activities is redundant), the final step of making a **well–formed** business process specification is required. The object of this step is to remove the redundancies and obtain a business process which is ready to be consumed. The final result is represented in Figure 2.11, (C).

The final business process actually comprises the details of both original processes, since by the means of OR–gate it becomes possible to follow either of the original processes. However, due to the nature of OR–gates, it becomes possible to follow *both* original processes simultaneously, which may be considered as an undesired side effect. Nevertheless, the implications of such side effects must be treated individually in each case and may be even beneficial since they may provide

additional opportunities.

Merging Based on a Change Set

Another approach to merge business process models is based on extraction of a **hierarchical change log**, which contains a set of atomic modifications which are needed in order to converts a business process A into a process B [Küster, Gerth, Förster and Engels 2008]. Such a representation makes it possible to perform an interactive business process merging, where a user picks one or more modifications from the list in order to apply them on one (or both) of the original processes.

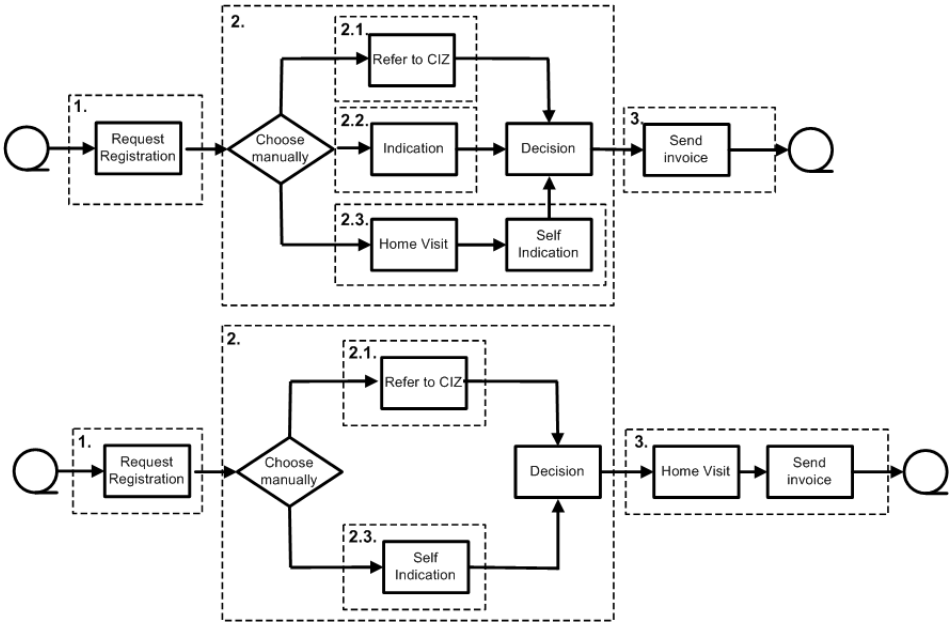


Figure 2.12: *Two business process models with SESE blocks highlighted.*

The principle of the extraction of a change log is the following. First, both original business process models are divided into so–called SESE blocks (single entry single exit, [Vanhatalo, Völzer and Leymann 2007]). In Figure 2.12, two business processes are illustrated, and for each of them dashed rectangles represent SESE blocks. Each of those block has its unique number (a number in the upper–left corner of each SESE block), and a one–to–one correspondence between SESE blocks of both processes is identified. In this example it is illustrated by the fact that the corresponding blocks bear the same cross–model number, and also note that the block number "2.2." of the process "A" has no counterpart in the process "B". In the

same way, a correspondence is established between the activities, in the example it is illustrated by the means of activity names, and the activity "Indication" has no counterpart.

Second, once the correspondence has been established, the atomic operations can be extracted. There are three kinds of modifications: (i) remove an element (an activity or a block), (ii) add an element, and (iii) move an element. In the last case, there are two sub–options: move within a block or move outside of the block. In the latter case, both source and target blocks must be mentioned. The list of modifications which lead from the process "A" into the process "B" is shown in Figure 2.13. The list is hierarchical, which means that some actions are independent and therefore can be made independently or in parallel. In the example above, there are two independent operations: move the activity "Home Visit" and remove the activity "Indication," while the latter operation consists of two steps. The order of removal steps is important, therefore they are listed sequentially.

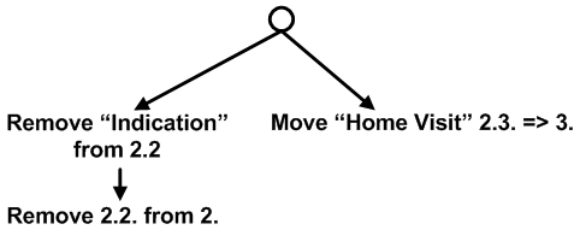


Figure 2.13: *The hierarchical change log.*

The way such operations are defined has a side effect that each of them is reversible, which means they can be either applied to the process "A" in order to make it equal to the process "B," or their mirror reflections can be applied to the process "B" in order to make it equal to the process "A." Also, a user may choose to apply selected change operations only, thus making possible to make a partial user–controlled merge of two business processes.

Theoretical Business Process Merging

A theoretical approach to business process merging is presented in [Sun, Kumar and Yen 2006]. A business process model (or a workflow, in the terms used by the authors) is formally represented in the form of Petri Nets. Then, for a fixed list of **workflow patterns** [van der Aalst, ter Hofstede, Kiepuszewski and Barros 2003], the authors derive a list of workflow merge patterns. The list of merge patterns is the following:

1. **Sequential Merge.** A pair of merging points is required in each of two workflows to be merged. For example, points p_1 and p_2 are in the first workflow, and p'_1 and p'_2 are in the second workflow. Then, there are two options. In the case of **replacement** merge, the part of the first workflow between the points p_1 and p_2 is replaced with the part of the second workflow between the points p'_1 and p'_2. The second case of **insertion** merge happens when $p_1 = p_2$, so the corresponding part of the second workflow is actually inserted into the first workflow in the place of point p_1.

2. **Parallel merge.** The situation is similar to the case of replacement sequential merge, but instead of replacing the corresponding part of the first worklow, and AND–split and corresponding AND–join are inserted into the first worklow model, and both of the original workflow parts co–exist in the result workflow as parallel branches of an AND–split.

3. **Conditional merge.** The situation is the same as the previous one, but instead of an AND–split an OR–split is used, thus providing a conditional split in the resulting workflow.

4. **Complex merge** happens in the case when there are more than one pair of merging points. In that case, each of such pairs is treated individually as described in the previous three options.

To conclude, in [Sun et al. 2006] the authors provide a theoretical foundation for the merging of business processes. However, the task of choosing the merge points is entirely up to the end user or the application framework which could possibly pick those merge points automatically, and that is the reason why this approach is mainly of theoretical interest.

2.2.2 Workflow Mining

An enterprise–level (CRM) system could in general be traced at the level of atomic events. Such events include invocations of some (Web–)services, sending notifications to the users, storing the information in a storage or in the Cloud and so on. The history of such events is called a **transaction log** and it essentially reflects the real lifecycle of a particular enterprise system. In the case of a Business Process Management System the information about the actual business process execution is stored and analysed as a part of a monitoring and diagnose stage, as illustrated in Figure 2.1.

Such information can be used in order to create a new business process based on the actual flow of events or to amend an existing business process model. The latter

situation happens when an existing business process model does not suit the actual customer's needs and therefore has to be deviated at run time.

A transaction log can be seen as a set or execution traces, and each of those traces consists of totally ordered set of individual events. It is presumed that there is a possibility to distinguish the individual events by their affiliation to a separate business process instance. Each execution trace in that case is a container of all events associated with an individual business process instance.

The task of **workflow mining** is the following: given a set of execution traces, create a model of a workflow which, being executed, would leave the same traces as contained in the original transaction log. The task of workflow mining has the affinity with the tasks of Business Intelligence (BI) and Data Mining (DM), and the solutions are provided in similar ways.

A theoretical approach to workflow mining was introduced in [van der Aalst and Dongen 2002]. There, each event in a transaction log considered to be a candidate for a single workflow task, and the name of the task is equal to the name of its corresponding event. Then, given a transaction log L is a set of execution traces $l_1, \ldots l_N \in L$, there are four types of interrelations between the activities a and b:

- $a > b$ iff there is a trace $l \in L$, such that for this trace the event a is located at i^{th} place and the event b is located at $(i+1)^{th}$ place, for some i;
- $a \rightarrow b$ iff $a > b$ and $b \not> a$;
- $a \# b$ iff $a \not> b$ and $b \not> a$;
- $a \parallel b$ iff $a > b$ and $b > a$.

To summarise, the first type means that in some cases a is immediately followed by b, the second type means that a is always followed by b, the third type means that a and b never appear together, and the fourth type means that a and b can appear in arbitrary order. The second type indicates that probably a is followed by b in some workflow model, while the third type might indicate that a and b are located on different branches of a XOR–split. The fourth type allows a and b to be located on parallel branches of an AND–split.

Having discovered the interrelations described above, a workflow model can be build, using the α–algorithm proposed in [van der Aalst and Dongen 2002]. The resulting workflow, though, may not cover the rare cases which were not appearing in the transaction log, and is not resilient to the noise which is a common issue in real–world systems.

In Table 2.2 (A), a few possible execution traces are listed in three separated columns. The order of events corresponds their temporal ordering, but the timestamps were omitted for the simplicity. In Table 2.2 (B), the interrelations between

different events are presented. Only the upper triangle is filled, because for any two given events their reverse relationship can be easily recovered. The final workflow in Petri–Net notation is illustrated in Figure 2.14.

A comprehensive survey of workflow mining is presented in [van der Aalst, van Dongen, Herbst, Maruster, Schimm and Weijters 2003].

Trace 1	Trace 2	Trace 3
1. RR	1. RR	1. RR
2. HV	2. CIZ	2. HV
3. SI	3. OR	3. SI
4. OR	4. DE	4. OR

(A)

	HV	CIZ	SI	OR	DE
RR	$\rightarrow$	$\rightarrow$	#	#	#
HV	—	#	$\rightarrow$	#	#
CIZ	—	—	#	$\rightarrow$	#
SI	—	—	—	$\rightarrow$	#
OR	—	—	—	—	$\rightarrow$

(B)

Table 2.2: *Three possible execution traces (A) and the interrelations implied by those traces (B).*

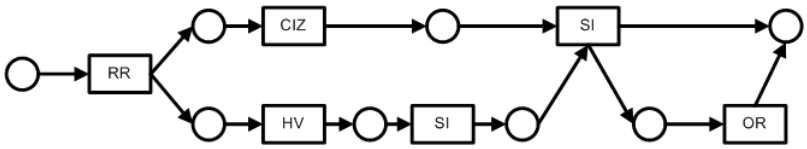


Figure 2.14: *The result of workflow mining.*

2.2.3 Other Approaches to Business Process Generation

The area of service–oriented computing provides with another option to build a business process via the usage of service composition techniques. A composition of services is generally a guideline on which services must be executed in order to fulfil a certain task.

Such a composition, once prepared, can be potentially reused several times, therefore, it can be treated as a business process. Consequently, the existing techniques for automated or semi–automated service compositions can be considered for the case of business process variability.

In general, a business process can be built of individual services basing on their implicit dependencies, which are driven by the input and output information of

each of the services [Eshuis and Grefen 2009, Brogi and Popescu 2006, Liang, Chakarapani, Su, Chikkamagalur and Lam 2004]. In result, the composition of services can be made in semi–automated way, because it may require the intervention of an end–user in order to build a concrete composition, as mentioned in [Eshuis and Grefen 2009].

Additionally, the advantages of integrating AI planning techniques for several applications in the field of Business Process Management have long been acknowledged. For instance, [Rodríguez-Moreno and Kearney 2002, Rodríguez-Moreno, Borrajo, Cesta and Oddi 2007, Madhusudan, Zhao and Marshall 2004] focus on how different planning approaches can assist at the business process definition phase, while the work presented in [Jarvis, Moore, Stader, Macintosh, Casson-du Mont and Chung 1999] investigates how planning can be used in case of domain state changes. In order to facilitate (semi-)automatic adaptation at runtime, AI planning techniques have been used from different viewpoints in the literature. In [Beckstein and Klausner 1999] the use of an intelligent assistant based on AI planning techniques is discussed, which can suggest compensation workflows or the re-execution of activities as a response to execution failures, with the help of meta-level knowledge incorporated in the workflow semantics.

In [Ferreira and Ferreira 2006] the use of machine learning is proposed in order to infer the preconditions and effects of activities provided the availability of a set of training examples, and then generate a partially ordered plan that complies to these rules. The framework aims at providing a candidate process that is able of achieving some business goals. At execution time, if an activity fails, an alternative candidate plan is provided.

The work closest to the problem of business process variability is the approach to BP adaptation through planning presented successively in [de Leoni, Mecella and De Giacomo 2007, de Leoni, De Giacomo, Lespèrance and Mecella 2009, Marrella and Mecella 2011]. This work uses several versions of Golog [Levesque, Reiter, Lespérance, Lin and Scherl 1997], which is based on planning by means of the situation calculus [Reiter 2001], to adapt a running process in case mismatches between the environment and the internal system representation are detected. In Golog the goal to be achieved has to be specified in a procedural way, as a non-deterministic program, as opposed to the use of high–level declarative goals, as the ones used by domain-independent planners, like the one presented herein. This implies that the adaptation process has to be pre–specified in an action-centric way, which requires domain-specific knowledge of the available services and hand–coding by a human expert. One advantage of the approach proposed in [Marrella and Mecella 2011] is that it can manage any unforeseen event, by continually comparing the environment with the expected outcomes according to the BP specification at each step of

execution.

The task of combining some actions in a dynamic way was studied in the field of semantic service composition by adopting AI planning techniques [Sohrabi and McIlraith 2010, Kaldeli, Lazovik and Aiello 2011, Au, Kuter and Nau 2005]. The common premise underlying these approaches is that services come along with semantic annotations that describe their behaviour in some convenient format, usually in terms of preconditions and effects. Many of the approaches proposed for service composition via automated planning, however, require that the set of supported solutions is pre–defined in some form of procedural templates, like in [Sohrabi and McIlraith 2010, Au et al. 2005].

Published as:
H. Groefsema and P. Bulanov and M. Aiello – *"Declarative Enhancement Framework for Business Processes,"* Int. Conference on Service-Oriented Computing (ICSOC–2011), LNCS 7084, pp. 495–504, 2011.
P. Bulanov and H. Groefsema and M. Aiello – *"Business Process Variability: A Tool for Declarative Template Design,"* Int. Conference on Service-Oriented Computing - Demo Track (ICSOC–2011), LNCS 7221, pp. 241–242, 2011.
H. Groefsema and P. Bulanov and M. Aiello – *"Imperative versus Declarative Process Variability: Why Choose?"* Submitted to IEEE Transactions on Software Engineering, 2012.

Chapter 3

The Case of Design Time

Among the different ways to address the problem of business process variability, the declarative one is the most promising in the terms of its flexibility and expressiveness [van der Aalst, ter Hofstede and Weske 2003, Schonenberg et al. 2008]. However, the introduction of declarative constraints poses additional difficulties while modeling a business process. Herein we introduce the Process Variability through Declarative and Imperative techniques (PVDI) framework which aims at allowing a high degree of process variability while preserving the main business goal of a process. PVDI accomplishes this goal through the combination of blueprinting and constraint techniques [Aiello, Bulanov and Groefsema 2010]. The hard to understand constraints are hidden from the end–user through easy to recognize graphical elements introduced to the blueprints or templates. Because of this, PVDI allows to benefit from both declarative and imperative variability modeling techniques.

Figure 3.1 describes the engineering process of PVDI. The process design part is shown on the left side and the template evolution part is shown on the right side. The dark activities represent the well-known BPM engineering steps, whereas the light activities are the ones which are specific to PVDI. Template design is driven by requirements, business goals, laws, regulations, etc. The template modeler uses this knowledge to model a template using traditional BPM techniques in combination with the graphical elements provided by PVDI. Once completed, constraints are generated from the PVDI graphical elements and embedded within the template. The template is then offered to process modelers as a resource. A process modeler may use this template to model a variant of the process using traditional BPM techniques, with the aid of the graphical PVDI elements which visually point the modeler to what is and what is not allowed. The resulting process is validated using the constraints specified within the template. If the validation returns faults,

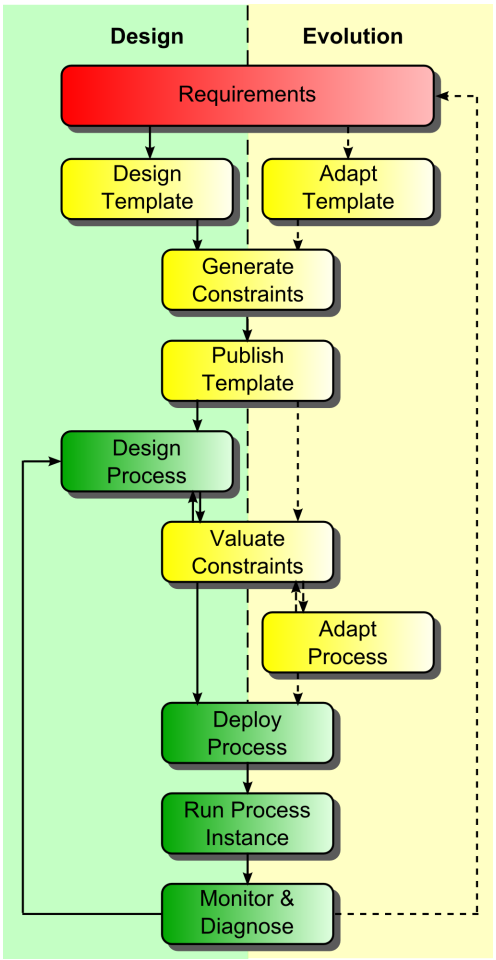


Figure 3.1: *PVDI Engineering Process.*

the graphical PVDI elements related to the faults are highlighted for easy reference. Once the modeler completes a valid variant, the variant may be deployed, executed, and monitored as a usual business process.

Whenever requirements drive towards template updates, the evolutionary cycle is entered. The template is updated, new constraints are generated, and the template is published again. At this point, whenever an existing variant based upon the previous version is run, a version check makes sure that the variant is up to date. If it is not, the variant is reevaluated with regards to the updated template. If this process returns faults, the variant must be adapted to adhere to the new ver-

sion of the template either automatically or manually. Once found correct, it may be redeployed, run, and monitored as before.

3.1 Basic Definitions

Before going into details of PVDI constraints specification, several basic concepts have to be defined. They include the formal definitions of processes, templates, variants, and other important aspects of PVDI.

A process is defined as a directed graph consisting of activities, gates, and events. Every activity contains multiple incoming and exactly one outgoing transition. The events, consisting of a single start and end event, contain exactly zero and multiple incoming, and one and zero outgoing transitions respectively. Gates on the other hand may contain multiple incoming and outgoing transitions. Since a process is defined as a directed graph, it can serve as a framework for a modal logic of processes, including computational tree logic$^+$ (CTL^+) (See Appendix A). Consider for example the process depicted in Figure 3.2. This process P consists of four activities A through D, a start event, an end event, and a gate.

Definition 3.1 (Process). *A* **process** P *is a tuple* $\langle S, T \rangle$ *where:*

- $S = (A \cup G \cup E)$ *is the set of activities, gates, and events;*
- $A = \{A_1 \dots A_n\}$ *is a finite set of activities;*
- $G = G_a \cup G_x$ *is a set of gateways, consisting of and– and xor–gates as defined by BPMN [Object Management Group (OMG) 2009];*
- $G = G_a \cup G_x$ *is a finite set of gateways;*
- E *is a set of events, containing a unique start* $\odot$ *and end* $\otimes$ *event;*
- $T \subseteq S \times S$ *is a binary relation on* S*;*
- $\forall s \in S : (s, \odot), (\otimes, s) \notin T$*;*
- $\forall a \in A \cup E$ *there is at most one* $s \in S$ *with* $(a, s) \in T$*;*
- $\forall s \in S \setminus \{\otimes\}$ *there is at least one* $s' \in S$ *with* $(s, s') \in T$*.*

Constraints are used by PVDI to capture restrictions on the variability offered within a PVDI template. These restrictions include, but are not limited to, process critical node and path information. A constraint in PVDI is a Computational Tree Logic$^+$ (CTL^+) formula. In order to use a process as a model for the CTL^+ constraints, we

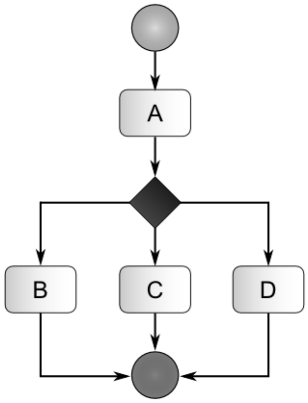


Figure 3.2: *An abstract process.*

introduce a set of variables and a valuation function. We use the **natural** valuation, that is, for each node $s \in S$ of a process we introduce a dedicated variable. The labeling function L is then built in such a way that each such dedicated variable is valuated to *true* for its corresponding node only. Additionally, under the natural valuation we can use the same letter to represent both activity and its corresponding variable. When evaluated, every constraint must valuate to *true* at every node $s \in S$ of a process.

Definition 3.2 (Constraint)**.** *A* **constraint** ϕ *over a process* P *is a computation tree logic*$^+$ (CTL^+) *formula whose propositional variables are in* $L(S)$ *of* P*, where* L *is a labeling function using the natural valuation.*

From a notation point of view, we use Φ to denote the set of constraints related to process P. A constraint is valid for a process P iff it is evaluated to *true* in every node of the process under the natural valuation. More formally,

Definition 3.3 (Constraint validity)**.** *Let* ϕ *be a constraint,* $\mathcal{M}$ *be a model built on the process* P *using the natural valuation, and* S *be the set of nodes of the process* P*. Then* ϕ *is* valid *iff* $\forall s \in S : \mathcal{M}, s \models \phi$*.*

Templates are used in PVDI as the basis for forming variants. Informally, a template is a process including constraints. Taking advantage of these formulations, we note that is possible to define underspecified processes. That is, a template may range from being a fully specified process as defined in Definition 4.1 to a set of nodes S with $T = \emptyset$. Since a template is not a process, any constraints $\phi \in \Phi$ within a template do not have to valuate to *true* for that template but only for its variants.

Definition 3.4 (Template). *A* **template** R *is a tuple* $\langle S, T, \Phi \rangle$

- S *as in definition 4.1;*
- $T \subseteq S \times S$ *is a binary relation on* S*;*
- $\forall s \in S : (s, \odot), (\otimes, s) \notin T$*;*
- $\forall a \in A \cup E$ *there is at most one* $s \in S$ *with* $(a, s) \in T$*.*

Variants in PVDI are processes which are based on a template and for which all constraints $\phi \in \Phi$ of that template are valid. A process based on a template for which not every constraint $\phi \in \Phi$ is evaluated to *true* at every node $s \in S$ of the process is therefore considered not to be a variant. Note that the sets of states and transitions of a template and its variant require no direct relation. This relation, instead, is implied through the constraints contained in the template and how they are required to valuate to true at every node of the variant. When adding additional constraints to define templates, the set of possible variants is reduced.

Definition 3.5 (Variant). *A* **variant** V *of a template* $R = \langle S_R, T_R, \Phi_R \rangle$ *is a process* $P = \langle S_V, T_V \rangle$ *such that* Φ_R *is valid for the process* P*.*

To ease the definition of constraints, we introduce the notion of a group in PVDI. That is a shorthand to describe a constraint spanning over a set of related nodes, in other words, to quantify a constraint over several nodes with different names.

Definition 3.6 (Group). *A* **group** G *in the process* $P = \langle S_P, T_P \rangle$ *is a nonempty subset of the set of nodes* S_P *of the process* P*. When a group* s_g *is used as input for a constraint instead of a single node* s*, then all occurrences of that single node* s *in the* CTL^+ *formula of that constraint are replaced by* $(s_1 \vee \ldots \vee s_n)$ *where* $s_1 \ldots s_n$ *are elements of the group* s_g*.*

3.2 Template Design

In order to support template design, PVDI expands the graphical language of the Business Process Modeling Notation (BPMN) [Object Management Group (OMG) 2009] with PVDI template elements. Each element consists of two parts: a *graphical element* which extends BPMN and a *constructive definition* which explains how to translate the graphical element into a constraint described in CTL^+. Template design in PVDI consists of several steps, illustrated in Figure 3.3. The dark steps are mandatory, and the light step is optional. Template design is naturally driven by requirements, such as those on the selection of activities, the order of activities,

the importance of an entire sub-process, etcetera. Using this information, the template modeler selects a set of activities for the template. Then, any transition may be added to the template. These transitions are largely optional, and may be used to guide the modeling of variants. At the same time, PVDI elements are added to the template. These elements can be seen as being graphical representations of constraints and include, but are not limited to, mandatory selection and ordering between activities. Next, the actual CTL^+ constraints are generated from the PVDI elements introduced in the previous step and embedded in the template. Finally, the template is published for use as a source for variants.

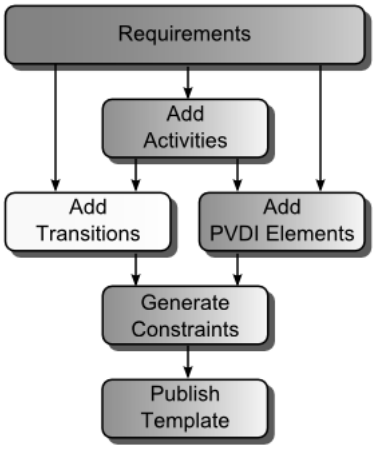


Figure 3.3: *Template creation.*

Let us consider these PVDI graphical elements and their translation into CTL^+ constraints one by one. The elements described here are not a final set and may by extended according to modeling needs. Since PVDI is in its design a declarative approach, first a number of elements using declarative techniques are discussed. Then, a number of elements more familiar to imperative techniques are introduced using PVDI's declarative approach.

3.2.1 Declarative Techniques

Declarative techniques are process flexibility techniques which focus on what tasks are performed [Schonenberg et al. 2008]. One inherent property of declarative techniques is that every variation is allowed, except for what is specifically disallowed. Because of this, declarative techniques are considered the more flexible of the approaches, but less strictly defined. Here we define four declarative PVDI elements, their graphical representations, and their translation to CTL^+ constraints. The ele-

ments discussed here are Mandatory Selection, Mandatory Execution, Ordered Execution, and Parallel/Exclusive Execution.

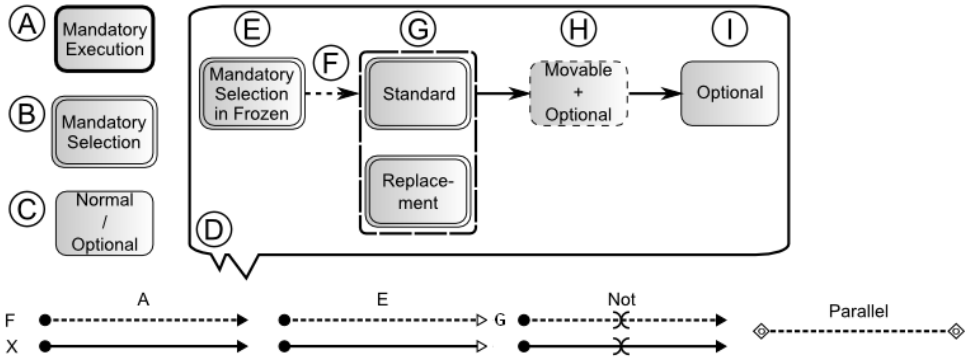


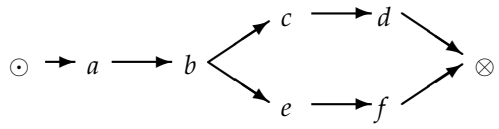
Figure 3.4: *PVDI Graphical Elements.*

Mandatory Between

The mandatory between constraint gives the option of marking nodes within a template in such a way that they must occur between two other nodes. Since mandatory between is used solely as a building block for other constraints, it does not have a corresponding graphical element. Mandatory between constrains the process using the CTL^+ formula $b \Rightarrow E([\neg e\ U\ s] \wedge Fe)$, meaning that at the begin node (b) there exists (E) a path for which we do not ($\neg$) encounter the end node (e) until (U) we encounter the marked node (s) and ($\wedge$) where we finally (F) encounter the end node (e). In other words, at b there is a path for which s comes before e. When a group is used instead of a single node, one of the nodes in the group must be encountered between b and e instead.

Definition 3.7 (Mandatory Between)**.** *The* **mandatory between** *is a constraint* $\alpha(b, e, s) = (b \Rightarrow E([\neg e\ U\ s] \wedge Fe))$.

Example 3.1 (Mandatory Between)**.** *In the figure below an example of a* **mandatory between** *constraint is illustrated. This constraint states that the node "b" must lie between the nodes "⊙" and "c," and this is indeed true, since there is a path from "⊙" to "c" which contains the node "b."*



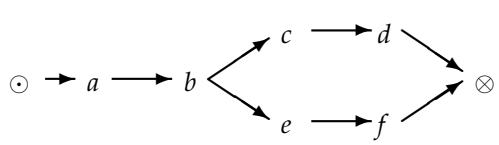
$$\alpha(\odot, c, b) \equiv \odot \Rightarrow E([\neg c\, U\, b] \wedge Fc).$$

Always Between

The always between constraint gives the option of marking nodes within a template in such a way that they must always occur in every path between two other nodes. Just like mandatory between, always between is used solely as a building block for other constraints, and therefore does not have a corresponding graphical element. Always between constrains the process using the CTL^+ formula $b \Rightarrow A([\neg e\ U\ s] \wedge Fe)$, meaning that at the begin node (b) for all (A) paths we do not ($\neg$) encounter the end node (e) until (U) we encounter the marked node (s) and ($\wedge$) we finally (F) encounter the end node (e). In other words, for all paths from b, s comes before e. When a group is used instead of a single node, one of the nodes in the group must always be encountered between b and e instead.

Definition 3.8 (Always Between)**.** *The* **always between** *is a constraint* $\beta(b, e, s) = (b \Rightarrow A([\neg e\ U\ s] \wedge Fe))$.

Example 3.2 (Always Between)**.** *In the figure below an example of an* **always between** *constraint is illustrated. This constraint states that the node "a" must always lie between the nodes "⊙" and "b," and this is indeed true, since there is only one path from "⊙" to "b," and this path contains the node "a." On the contrary, the constraint* $\beta(\odot, c, b)$ *is not valid, since not all paths starting from* $\odot$ *go through the node "c."*



$$\beta(\odot, b, a) \equiv \odot \Rightarrow A([\neg b\, U\, a] \wedge Fb).$$

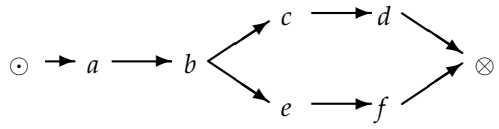
Not Between

The not between constraint gives the option of marking nodes within a template in such a way that they must never occur in any path between two other nodes. Just like the others, not between is used solely as a building block for other constraints, and therefore does not have a corresponding graphical element. Not between constrains the process using the CTL^+ formula $b \Rightarrow A[\neg s\ U\ e]$, meaning that at the

begin node (b) for all (A) paths we do not ($\neg$) encounter the marked node (s) until (U) we encounter the end node (e). In other words, for all paths from b, we do not encounter s before e. When a group is used instead of a single node, none of the nodes in the group may be encountered between b and e instead.

Definition 3.9 (Not Between)**.** *The* **not between** *is a constraint* $\gamma(b, e, s) = (b \Rightarrow A([\neg s\ U\ e] \vee G\neg e))$.

Example 3.3 (Not Between)**.** *In the figure below an example of a* **not between** *constraint is illustrated. This constraint states that the node "f" must not lie between the nodes "*$\odot$*" and "b," and this is indeed true, since the node "f" lies after the node "b."*



$\gamma(\odot, b, f) \equiv \odot \Rightarrow A([\neg f\ U\ b] \vee G\neg b)$.

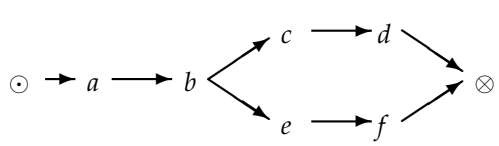
Mandatory Selection

The mandatory selection constraint gives the option of marking nodes within a template in such a way that they must be selected for use in a variant. Any node which is not marked as mandatory is therefore considered to be optional.

Definition 3.10 (Mandatory Selection)**.** *A* **mandatory selection** *is a constraint* $\phi(s)$ *such that* $\phi(s) = \alpha(\odot, \otimes, s)$*, as described in Definition 3.7.*

A mandatory selection consists of the mandatory between where the start $\odot$ and end $\otimes$ events are used as the begin and end nodes of the mandatory between. As a result, the marked node must at least occur in one path between start and end events. In case a group is used instead of a single node, a disjunction is formed between the nodes within the group as defined in Definition 3.6, resulting in at least one of these nodes to occur within a path between the start and end events. The graphical representation of the mandatory selection element can be seen in Figure 3.4 as B, E, and the activities within group G.

Example 3.4 (Mandatory Selection)**.** *In the figure below an example of a* **mandatory selection** *constraint is illustrated. This constraint states that the node "a" must always be selected, or, in other words, must lie on at least one of the paths from* $\odot$ *to* $\otimes$*.*



$\phi(a) \equiv \alpha(\odot, \otimes, a).$

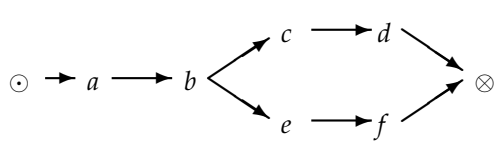
Mandatory Inclusion

The mandatory inclusion element allows us to mark a node within a template as mandatory for execution path in every run–time instance of every variant. In other words, every path must contain the marked node such that any execution path taken in a run–time instance of the variant encounters the marked node.

Definition 3.11 (Mandatory Inclusion)**.** *The* **mandatory inclusion** *is a constraint* $\phi(s)$ *such that* $\phi(s) = \beta(\odot, \otimes, s)$*, as described in Definition 3.8.*

A mandatory inclusion consists of the always between where the start $\odot$ and end $\otimes$ events are used as the begin and end nodes of the always between. As a result, all paths between the start and end events must include the marked node. When a group is used instead of a single node, at least one of the nodes in the group must occur in each path. Mandatory inclusion can thus be seen as a stricter version of mandatory selection (Definition 3.10). The graphical representation of the mandatory execution element can be seen in Figure 3.4 as A.

Example 3.5 (Mandatory Inclusion)**.** *In the figure below an example of a* **mandatory inclusion** *constraint is illustrated. This constraint states that the node "a" must always be executed, or, in other words, must lie on each of the paths from* $\odot$ *to* $\otimes$*. Note that the constraint* $\phi(c)$ *is not valid, since the only one of two possible paths from* $\odot$ *to* $\otimes$ *contains the node "c."*



$\phi(a) \equiv \beta(\odot, \otimes, a).$

Ordered Execution

The ordered execution element allows to define the order of nodes in the template when used in the paths of the variants. An ordered execution is a relation between two nodes $p \in S$ and $q \in S$ stating the relative order of these two nodes in one

or all execution paths. Groups of nodes may be used for both p and q, in which case *every* element within p must, or must not, be followed by *any* element within q. Note that neither p nor q becomes mandatory through the ordered execution. However, when p is included q could become mandatory as a result. The graphical representation of the ordered execution elements can be seen as flows at the bottom of Figure 3.4. Here, ordered execution is described over two dimensions; path and distance. The rows relate to the temporal dimensions; F (Finally) and X (neXt), which require the linked elements to either follow each other eventually or immediately. The first two columns relate to the paths; E (there Exists a path) and A (for All paths), which require the linked elements to follow each other in either a path or all paths respectively. The third column represents a negation of two of these flows. Note that the negation of an ordered execution representing F results in the use of a G (Globally) instead for desired result. In Table 3.1, the corresponding CTL^+ formulas are displayed for each of the mentioned constraints. Here, the rows correspond to the CTL path quantifiers, and the columns to the CTL state quantifiers plus the optional negation. The formulas in the first row define a relation between p and q where q should Finally follow p in all paths, a path, no path, and not a path respectively. The second row defines the same relations but only for the neXt node instead.

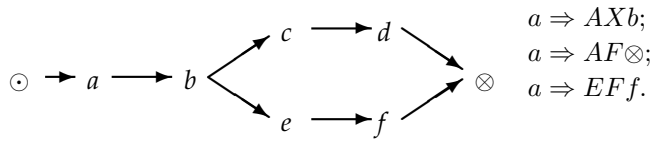
Definition 3.12 (Ordered Execution)**.** *An* **ordered execution** *is a constraint* $\phi(p, q, \Omega, \Pi) = (p \Rightarrow \Omega\Pi q)$*, with:*

- p, q *are nodes* $s \in S$ *or non-overlapping groups;*
- $\Omega \in \{A, E\}$ *is a state quantifier;*
- $\Pi \in \{X, \neg X, F, \neg F\}$ *is a path quantifier.*

$\Omega \backslash \Pi$	X	$\neg X$	F	$\neg F$
A	$p \Rightarrow AXq$	$p \Rightarrow AX\neg q$	$p \Rightarrow AFq$	$p \Rightarrow AG\neg q$
E	$p \Rightarrow EXq$	$p \Rightarrow EX\neg q$	$p \Rightarrow EFq$	$p \Rightarrow EG\neg q$

Table 3.1: *Possible* Ordered Execution*s*

Example 3.6 (Mandatory Inclusion)**.** *In the figure below several examples of different options of* **ordered execution** *are illustrated.*



Parallel/Exclusive Execution

The parallel/exclusive execution element allows to enforce the non occurrence of a given two nodes in the same path . Parallel/exclusive execution constrains the process in such a way that from two nodes $p \in S$ and $q \in S$ all paths (A) globally (G) may not encounter the other node. The result is that the only one of the nodes p and q may be selected to be used within a variant, or that they both must be preceded by a XOR– or AND–gate.

Definition 3.13 (Parallel/Exclusive execution)**.** *A* **parallel/exclusive execution** *is a constraint* $\phi(p, q) = (p \Rightarrow AG\neg q) \wedge (q \Rightarrow AG\neg p)$*, where* p, q *are distinct nodes* $s \in S$ *or non-overlapping groups.*

Note that any required precedence of a specific gate type should be constrained by the other means (ordered execution). When groups are used for p and/or q instead of a single node, every node within p must not be followed by any node within q, and vice versa. The graphical representation of the parallel/exclusive execution element can be seen as a flow at the bottom right of Figure 3.4.

3.2.2 Imperative Techniques

Imperative techniques differ from declarative techniques by allowing no variations except for those which are specified beforehand. Because of this, imperative techniques are considered less flexible, but allow for an easy to use straightforward design process for variants. Given the nature of these techniques, we provide first two "areas" representing imperative specifications and then present several *modifications*, which change the flexibility of these areas in order to allow for variation points, that is, elements of a business process where change may occur to support imperative variability [Aiello et al. 2010].

Closed Area

A closed area constrains the selected area in such a way that every node becomes mandatory to select (Definition 3.10) and no nodes other than those already in the area may be introduced to it. Closed areas allow exactly one incoming and exactly one outgoing flow to and from it.

Definition 3.14 (Closed Area)**.** *A* **closed area** *C over a group G is a set of constraints built over the set of activities $G_A = \{a_1, a_2, \ldots a_n\}$ of the group G and two dedicated start $\odot_C$ and end $\otimes_C$ nodes of the group. The set of constraints consists of the following:*

- Mandatory between *constraints $\alpha(\odot_C, \otimes_C, a_i)$ (Definition 3.7) for each activity $a_i \in G_A$;*
- *For the group $G_A^{-1} = S \setminus G_A$:*
  - $G_A^{-1} \Rightarrow EX\neg(a_1 \vee a_2 \vee \ldots \vee a_n \vee \otimes_C)$;
  - $(a_1 \vee a_2 \vee \ldots \vee a_n \vee \odot_C) \Rightarrow EX\neg G_A^{-1}$;
- *A* closed *constraint described by the CTL^+ formula $\odot_C \Rightarrow A[(\odot \vee a_1 \vee a_2 \vee \ldots \vee a_n)\ W\ \otimes_C]$, where $a_1, a_2, \ldots a_n$ are members of the set G_A.*

Frozen Area

Frozen areas specify areas in a template which may not be altered when designing variants unless specifically allowed. A frozen area constrains a part of a template in such a way that every node becomes mandatory to select (Definition 3.10) and that every path from every node allows for no variation until the exit of the area. Effectively, every path between its start and end is "frozen" and may not be changed. A frozen area is defined over a group (Definition 3.6) which may be used as such with regard to any other techniques described herein. The graphical representation of the frozen area can be seen in Figure 3.4 as D.

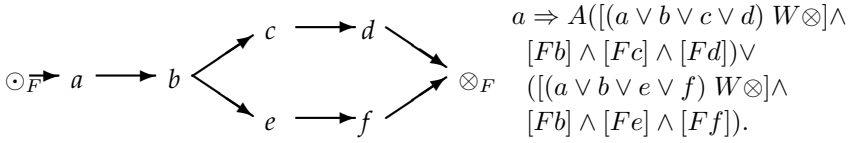
Definition 3.15 (Frozen area)**.** *A* **frozen area** *F over a group G is a set of constraints built over the set of activities $G_A = \{a_1, a_2, \ldots a_n\}$ of the group G and two dedicated start $\odot_F$ and end $\otimes_F$ events of the group. The set of constraints consists of the following:*

- Mandatory between *constraints $\alpha(\odot_F, \otimes_F, a_i)$ (Definition 3.7) for each activity $a_i \in G_A$;*
- *For the group $G_A^{-1} = S \setminus G_A$:*
  - $G_A^{-1} \Rightarrow EX\neg(a_1 \vee a_2 \vee \ldots \vee a_n \vee \otimes_F)$;
  - $(a_1 \vee a_2 \vee \ldots \vee a_n \vee \odot_F) \Rightarrow EX\neg G_A^{-1}$;
- Path *constraints described by the CTL^+ formula $a_i \Rightarrow A(\pi_i^1 \vee \ldots \vee \pi_1^T)$ for each activity $a_i \in G_A \bigcup \{\odot_F\}$, where:*
  - *each of the sub–formulas π_i^j corresponds to a single path p_i^j leading from the activity a_i to the end of the group $\otimes_F$. There are as many sub–formulas π_i^j as there are distinct paths leading from a_i to the end;*

– *for each of such paths* $p_i^j = \{a_i, p_1^j, \ldots p_k^j, \otimes_F\}$*, the corresponding sub–formula* π_i^j *is described by:* $\pi_i^j = [(a_i \vee p_1^j \vee \ldots \vee p_k^j)\ W\ \otimes_F] \wedge [F\ p_1^j] \wedge \ldots \wedge [F\ p_k^j]$*;*

– *for the case of a simple path* $p_i^j = \{a_i, \otimes_F\}$*, the sub–formula* π_i^j *is reduced to* $\pi_i^j = X\ \otimes_F$*.*

Example 3.7 (Frozen Area)**.**



Assume the process illustrated above is encoded as a set of CTL^+ *formulas. The formulas for* $\odot$*,* a*,* b *are the longest because there exist two possible paths from them to* $\otimes$*. Therefore, according to Definition 3.15, the formula splits into two pieces, one for each path. Consider the upper branch of the process, and the path* $p_a^1 = \{a, b, c, d, \otimes_F\}$*. The formula* π_a^1 *for that path is therefore the following:* $[(a \vee b \vee c \vee d)\ W\ \otimes_F] \wedge [Fb] \wedge [Fc] \wedge [Fd]$*. The same applies to the bottom branch. The resulting path–preserving formula for the node "a" is illustrated in the figure above.*

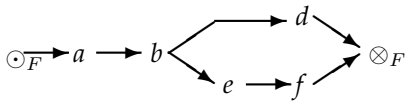
Optional Nodes

Allowing nodes to be optional is a modification of the definition of a frozen (Definition 3.15) area which allows for the removal of otherwise mandatory (Definition 3.10) nodes constrained by an area. Allowing for optional nodes modifies the constraints of the area in such a way that the affected nodes are no longer mandatory to select or may be bypassed within the area and thus become optional. The graphical representation of the optional nodes within a area can be seen in Figure 3.4 at H and I.

Modification 1 (*Allow for Optional Node*). Allowing for an optional node s is accomplished through the following modifications on the constraints:

- The **mandatory selection** constraint $\phi(\odot_F, \otimes_F, s)$ corresponding to the activity s is removed;
- All of the path sub–formulas π_i^j are modified in the following way: if the clause $[F\ s]$ is a part of the formula π_i^j, then that clause is removed from the formula π_i^j.

Example 3.8 (Frozen Area — Node Removal)**.**



$$a \Rightarrow A([(a \vee b \vee c \vee d)\ W\ \otimes_F] \wedge [Fb] \wedge [Fd]) \vee ([(a \vee b \vee e \vee f)\ W\ \otimes_F] \wedge [Fb] \wedge [Fe] \wedge [Ff]).$$

Consider the same process as illustrated in Example 3.7, and allow the node "c" to be optional. It implies the removal of appropriate "mandatory" constraint, and the structure–preserving formulas are modified to reflect the optional nature of the node "c." Namely, the clause $[Fc]$ *is removed, as it is show in the figure above.*

To illustrate how optional nodes work in practice, we removed the node "c" from the process, leaving the rest of the process intact. To check the corresponding formula, consider the path sub–formula $[(a \vee b \vee c \vee d)\ W\ \otimes_F]$*. It will be valid for the upper path of the process, since the lack of the node "c" does not violate the "until" formula. Also, the rest of the formula is also valid because of the absence of the clause* $[Fc]$*.*

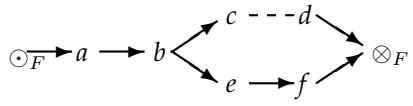
Inserting Nodes

Allow node insertion is a modification of the constraints of a frozen area (Definition 3.15) which allows for the insertion of nodes at specific spots within a path constrained by a frozen area. The graphical representation of the option to insert nodes at spots within a frozen area can be seen in Figure 3.4 as the arrow at F.

Modification 2 (*Allow Node Insertion*). Allowing for the insertion of a node in a certain spot which lies between the activities a_t and a_{t+1} is accomplished with the following modifications:

- for all paths $p_i^j = \{a_i, a_{i+1} \ldots a_t, a_{t+1}, \ldots \otimes_F\}$ which contain both a_t and a_{t+1}, the corresponding path formula π_i^j is modified in the following way:
- $\pi_i^j = [(a_i \vee a_{i+1} \vee \ldots \vee a_t)\ U\ AF\ \psi] \wedge [Fa_{i+1}] \ldots \wedge [Fa_t]$, where ψ is the path–preserving formula for the activity a_{t+1} according to the definition of frozen area (Definition 3.15);
- note that the formula ψ may be in turn modified because of the existence of more places which allow for node insertion.

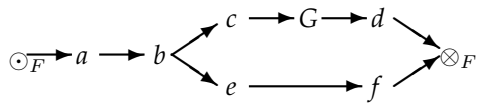
Example 3.9 (Frozen Area — Insert a Node)**.**



$a \Rightarrow A([(a \vee b \vee c)\ W\ (AF\ \psi)] \wedge$
$[Fb] \wedge [Fc] \wedge [Fd]) \vee \dots)$
$\psi = A[(d\ W\ \otimes_F)].$

Consider the same process as illustrated in Example 3.7, and allow an insertion point between the nodes "c" and "d," which is illustrated with a dashed line. For the simplicity, we show only the first part of the formula which covers the upper branch of the process, since the second part of the formula remains intact.

In the figure below, one of the possible process modification is shown, with a new node "G" inserted between the nodes "c" and "d."



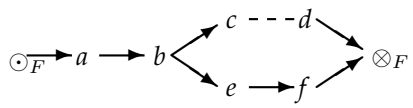
Moving Nodes

Allow node movement is a modification of the constraints of a frozen area (Definition 3.15) which allows a node to be moved within that area. Note that allowing a node to be moved does not automatically mean that it has a place to be moved to. Nor does it make the node optional. The graphical representation of the option to move nodes within a frozen area can be seen in Figure 3.4 at H.

Modification 3 (*Allow Node Movement*). Allowing for the moving of a node s is accomplished with the following modifications:

- the path–preserving formula for the node s of the type $s \Rightarrow \phi$ is removed;
- all of the other path sub–formulas π_i^j are modified in the following way: if the clause $[F\ s]$ is a part of the formula π_i^j, then that clause is removed from the formula π_i^j.

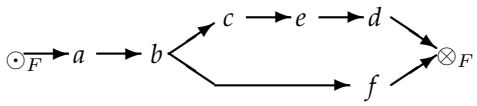
Example 3.10 (Frozen Area — Move a Node)**.**



$a \Rightarrow A([(a \vee b \vee c)\ W\ (AF\ \psi)] \wedge$
$[Fb] \wedge [Fc] \wedge [Fd]) \vee$
$([(a \vee b \vee e \vee f)\ W \otimes] \wedge [Fb] \wedge [Ff]);$
$\psi = A[(d\ W\ \otimes_F)].$

Consider the same process as illustrated in Example 3.7, and let us allow the activity "e" to be moved. However, just making a node "movable" does not help, since there is no potential target place for such a traveling node, therefore, we will also allow for node insertion between the nodes "c" and "d," as already described in Example 3.9. In result, the figure above shows

In the figure below, we explored the possibility to move the node "e," and we moved it into the only admissible place (apart from its original home place), which is the place between the nodes "c" and "d."



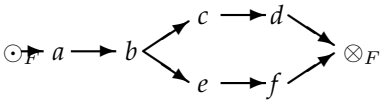
Replacing Nodes

Allow node replacement is a modification of the constraints of a frozen area (Definition 3.15) which allows the choice to include one of several nodes at a certain point in a path of a frozen area. The graphical representation of the option to replace nodes within a frozen area can be seen in Figure 3.4 at G.

Modification 4 (*Allow Node Replacement*). Allowing for the replacement of node s_j^0 with nodes $s_j^1, \ldots, s_j^i$ in a path is accomplished through the following modification of mandatory and path–preserving constraints:

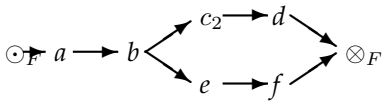
- Replace every occurrence of s_j^0 in every constraint with the following clause: $(s_j^0 \vee s_j^1 \vee \ldots \vee s_j^i)$.

Example 3.11 (Frozen Area — Replate a Node)**.**



$$
\begin{aligned}
a \Rightarrow A([(a \vee b \vee (c \vee c_1 \vee c_2) \vee d)\ W \otimes] \wedge \\
[Fb] \wedge [F(c \vee c_1 \vee c_2)] \wedge [Fd]) \vee \\
([(a \vee b \vee e \vee f)\ W \otimes] \wedge \\
[Fb] \wedge [Fe] \wedge [Ff]).
\end{aligned}
$$

Consider the same process as illustrated in Example 3.7, and let us allow to replace the activity "c" with either "c_1" or "c_2." The figure above illustrate the necessary modifications in the path–preserving formula. In result, the process which contains the activity "c_2" in place of the activity "c" is still valid.



3.3 Constraint Relations

So far we have introduced constraints as CTL^+ implications. Although these offer a considerable amount of expressivity for variability, we need more complex constructions to capture other important variability functionality. COVAMOF for example allows relations between different variation points at different levels of abstraction. These so called realization relations *"specify rules that determine which variants or values at variation points at lower levels should be selected in order to realize the choice at variation points at higher levels"* [Sinnema et al. 2006]. Although PVDI does not feature variation points, a similar mechanism is supported in the form of Constraint Relations. A constraint relation is a constraint which forms a zeroth–order logic formula over two other constraints, that is, a formula without quantifiers. Next, we introduce a number of constraint relations.

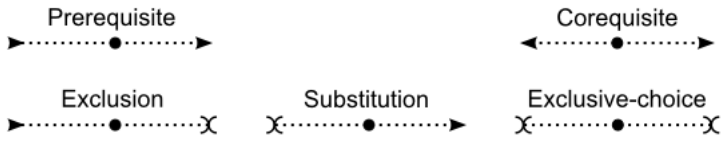


Figure 3.5: *PVDI Graphical Elements for Variation Relations.*

3.3.1 Prerequisite

The prerequisite constraint relation defines an affiliation between two nodes regarding their inclusion, but without any restrictions on ordering. The prerequisite relation constrains the process in such a way that if p is included then q must be included as well. When a group is used for p or q, then when at least one node from the set p is included, then at least one node from the set q must be included as well. The graphical representation of the prerequisite relation can be seen in Figure 3.5 at the top left.

Definition 3.16 (Prerequisite)**.** *A* **prerequisite** *constraint* $\phi(p,q) = (\alpha(\odot,\otimes,p) \Rightarrow \alpha(\odot,\otimes,q))$*, as described in Definition 3.7, with* p,q *being different nodes* $s \in S$ *or non overlapping groups.*

3.3.2 Exclusion

The exclusion constraint relation defines an affiliation between two nodes regarding the exclusion of one of them. The exclusion relation constrains the process in such a way that if p is included then q must not be included. When a group is used for p or q, then when at least one node from the set p is included, then no node from the set q may be included. The graphical representation of the exclusion relation can be seen in Figure 3.5 at the bottom left.

Definition 3.17 (Exclusion)**.** *An* **exclusion** *constraint* $\phi(p, q) = (\alpha(\odot, \otimes, p) \Rightarrow \gamma(\odot, \otimes, q))$*, as described in Definition 3.7 and Definition 3.9, with* p, q *being different nodes* $s \in S$ *or non overlapping groups.*

3.3.3 Substitution

The substitution constraint relation defines an affiliation between two nodes regarding their substitution. The substitution relation constrains the process in such a way that if p is not included then q must be included instead. When a group is used for p or q, then when no node from the set p is included, then at least one node from the set q must be included. The graphical representation of the substitution relation can be seen in Figure 3.5 at the bottom in the middle.

Definition 3.18 (Substitution)**.** *A* **substitution** *constraint* $\phi(p, q) = (\gamma(\odot, \otimes, p) \Rightarrow \alpha(\odot, \otimes, q))$*, as described in Definition 3.7 and Definition 3.9, with* p, q *being different nodes* $s \in S$ *or non overlapping groups.*

3.3.4 Corequisite

The corequisite constraint relation defines an affiliation between two nodes regarding their inclusion. The corequisite relation constrains the process in such a way that if p is included q then must be included as well, and vice versa. When a group is used for p or q, then when at least one node from the set p is included, then at least one node from the set q must be included as well, and vice versa. The graphical representation of the corequisite relation can be seen in Figure 3.5 at the top right.

Definition 3.19 (Corequisite)**.** *A* **corequisite** *constraint* $\phi(p, q) = \psi(p, q) \wedge \psi(q, p)$*, where* ψ *is the* prerequisite *constraint (Definition 3.16) and* p, q *are different nodes* $s \in S$ *or non overlapping groups.*

3.3.5 Exclusive–Choice

The exclusive–choice constraint relation defines an affiliation between two nodes regarding their inclusion and exclusion. The exclusive–choice relation constrains the process in such a way that if p is included then q must not be included, and vice versa. When a group is used for p or q, then when at least one node from the set p is included, then no node from the set q may be included, and vice versa. The graphical representation of the exclusive–choice relation can be seen in Figure 3.5 at the bottom right.

Definition 3.20 (Exclusive–Choice)**.** *An* **exclusive–choice** *constraint* $\phi(p,q) = \psi(p,q) \wedge \psi(q,p)$*, where* ψ *is the* exclusion *constraint (Definition 3.17) and* p, q *are different nodes* $s \in S$ *or non overlapping groups.*

3.4 Variant Design: An Example

The design of variants in PVDI is naturally eased by the PVDI graphical elements. Every element introduced in the previous sections guides the design towards a set of possible variants. Take for example the PVDI template depicted in Figure 3.6. This template specifies a simple room reservation process including three different rooms: practical labs, classrooms, and meeting rooms. Because the three rooms are grouped, and an ordered execution constraint specifies that the start element must be followed by the group, at least one of those three room types must be included, and more than one room may be included. The group is then followed by a frozen area comprising of four activities: "Lock table," "View rooms," "Reserve room," and "Error." Of these, two activities are mandatory to select, and two ("Lock table" and "Error") are allowed to be removed. However, by using an exclusive–choice constraint relation we specify that at least one of the two optional activities must be included. In this way, we force either a lock table before reserving mechanism, or a first come first serve mechanism including a success/error report in case of failure when the room was already reserved by somebody else.

Activity	Variable	Activity	Variable
Start	s	Group end	e_g
End	e	Lock	lo
Practical Lab	pr	View Rooms	vr
Classroom	cr	Reserve Rooms	rr
Meeting room	mr	Check Error	ce
Group start	s_g		

Table 3.2: *PVDI Room Reservation Elements.*

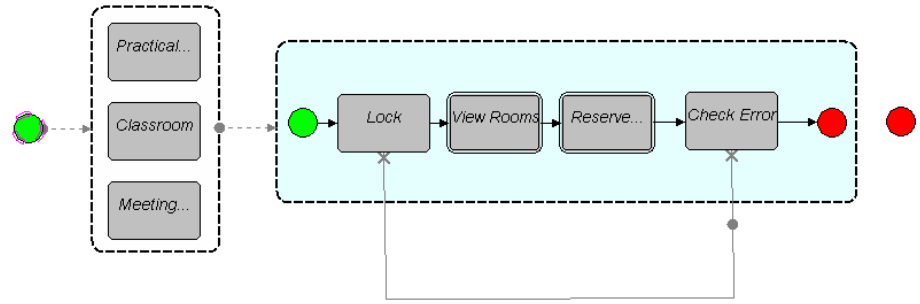


Figure 3.6: *PVDI Room Reservation Template.*

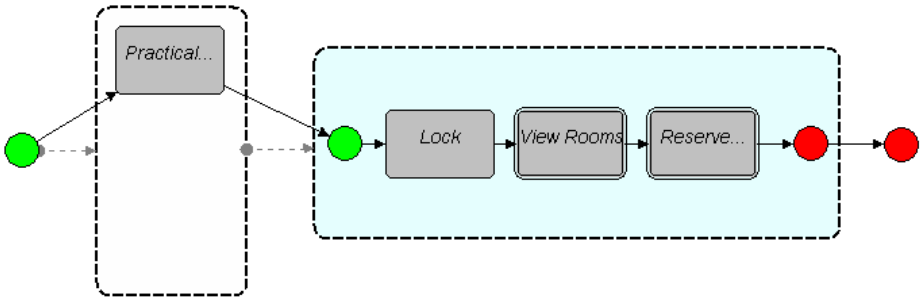


Figure 3.7: *A Valid Variant of Room Reservation Process.*

Once a template is created, its PVDI elements are automatically converted by the tool to a set of CTL^+ constraints as described in sections 3.2 and 3.3. The following constraints are generated from the PVDI template depicted in Figure 3.6. We use shorthand notations for each element, the meaning of which can be found in Table 3.2.

In the equation below, each line represents a single constraint generated on the basis of the visual constraints in Figure 3.6. Lines 1 and 2 represent the flow constraints leading to and from the group, respectively. Lines 3 to 7 represent the path–preserving constraints for the frozen area. Lines 8 and 9 include the mandatory constraints for the two mandatory nodes in the area, and lines 10 and 11 list the restrictions on entering and exiting the area through its start and end nodes only. Finally, line 12 lists the exclusive–choice constraint.

Next, variants are designed using the template. An example of a valid busi-

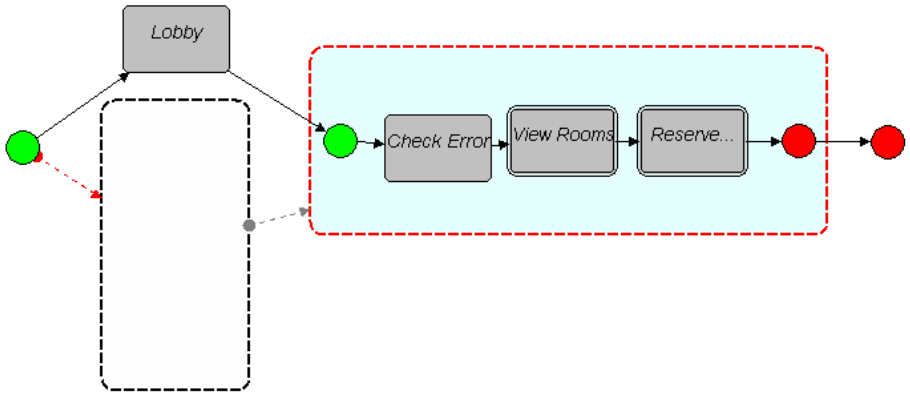


Figure 3.8: *An Example of Invalid Variant for Room Reservation Process.*

ness process variant is shown in Figure 3.7. There a practical lab is chosen and an exclusive–lock mechanism is used to avoid any collisions.

Figure 3.8 shows an erroneous example. Two PVDI elements are being highlighted by the tool, which indicates that their related formulas are being violated. First, none of three predefined room types is used, resulting in a violation of the constraint in line 1. Second, the frozen area structure is corrupted since the activity "Check Error" is placed before the activity "View Rooms", resulting in a violation of the constraint in line 7.

$$s \Rightarrow AF(pr \vee cr \vee mr) \tag{3.1}$$
$$(pr \vee cr \vee mr) \Rightarrow AF(s_g \vee lo \vee vr \vee rr \vee rr \vee ce \vee e_g) \tag{3.2}$$
$$s_g \Rightarrow A([(s_g \vee lo \vee vr \vee rr \vee ce)We_g] \wedge Fvr \wedge Frr) \tag{3.3}$$
$$lo \Rightarrow A([(lo \vee vr \vee rr \vee ce)We_g] \wedge Fvr \wedge Frr) \tag{3.4}$$
$$vr \Rightarrow A([(vr \vee rr \vee ce)We_g] \wedge Frr) \tag{3.5}$$
$$rr \Rightarrow A[(rr \vee cc)We_g] \tag{3.6}$$
$$ce \Rightarrow A[(ce)We_g] \tag{3.7}$$
$$s_g \Rightarrow E([\neg e_g U vr] \wedge Fe_g) \tag{3.8}$$
$$s_g \Rightarrow E([\neg e_g U rr] \wedge Fe_g) \tag{3.9}$$
$$[s \vee e \vee pr \vee cr \vee mr] \Rightarrow \neg EX[lo \vee vr \vee rr \vee ce \vee e_g] \tag{3.10}$$
$$[s_g \vee lo \vee vr \vee rr \vee ce] \Rightarrow \neg EX[s \vee e \vee pr \vee cr \vee mr] \tag{3.11}$$
$$[\alpha(s,e,lo) \Rightarrow \gamma(s,e,ce)] \wedge [\alpha(s,e,ce) \Rightarrow \gamma(s,e,lo)] \tag{3.12}$$

3.5 Process Healthiness

There is a number of possible metrics to verify if a business process model is healthy or not. In [Wynn, Verbeek, van der Aalst, ter Hofstede and Edmond 2009] three main characteristics of a healthy, or sound, processes are mentioned. Namely, the (weak) option to complete, proper completion , and the absence of "dead" transitions. In PVDI, those characteristics are ensured through the help of a healthiness constraint.

Definition 3.21 (Healthiness)**.** *The* **healthiness** *is a constraint* $\phi(s) = \alpha(\odot, \otimes, s)$*, as described in Definition 3.7.*

The healthiness constraint disallows dead–ends and ensures that all nodes are reachable at the same time (Definition 3.21). It consists of a mandatory between where the start $\odot$ and end $\otimes$ events are used as the begin and end nodes of the mandatory between. In contrast with the constraints discussed in previous sections, which are generated from the template before modeling a variant, the constraints described in this section are generated after modeling a variant and directly prior to validation. Although the constraint seems equal to the mandatory selection (Definition 3.10), the the difference in the time of generation allows for different uses. A process (Definition 4.1) is considered correct when all correctness requirements have been evaluated to *true* at all nodes of the process.

Definition 3.22 (Healthy Process)**.** *A process* P *is* **healthy** *iff* $\forall s \in S_p$ *the healthiness constraint* $\phi(s)$ *is valid at every node* $s \in S_P$ *of the process.*

3.6 Variant Validation

Processes are required to be validated after their derivation from a template. A process is *valid* with respect to a template if it is healthy (Definition 3.22) and if it is a variant (Definition 3.5) of this template.

Definition 3.23 (Valid Process)**.** *A process* P *is* **valid** *with respect to a template* R *iff it is a variant of* R *and is healthy.*

As specified in definition 3.5, a process P is a variant of a template R if the set of constraints Φ_R is valid for P. The same is true for healthiness (Definition 3.22), P is healthy if the set of healthiness constraints Φ_H is valid for P. In turn, according to Definition 3.3, these constraints are valid if $\forall s \in S_P : \mathcal{M}, s \models \Phi_R \cup \Phi_H$. As such, validation entails that the process is evaluated against these sets of constraints. We therefore propose an algorithm which valuates every constraint at every node for every path emerging from that node.

3.6.1 Model Conversion

Model checking is a technique used to automatically verify models against a given specification. In classical model checking (e.g., [Clarke, Grumberg and Peled 2000]), a model is defined as a finite state machine, and is checked against a set of formulas of propositional or modal logic. In case of PVDI, a process, which is defined as a directed graph (Definition 4.1), is validated against a set of constraints expressed as CTL^+ logic formulas. We therefore employ model checking techniques when verifying variants [Clarke, Emerson and Sistla 1986]. A variant is defined as a process $P = \langle S, T \rangle$ for which all constraints are valid. A CTL model $\mathcal{M} = \langle S, T, L \rangle$ consists of a set of states S, a set of transitions T, and a valuation function L [Emerson and Halpern 1985, Clarke et al. 1986]. In order to get the corresponding model $\mathcal{M}$ of P, we map S and T such that loops are mapped only once, but infinite traversals of loops are avoided. Since we evaluate business processes which at most might be long living, but never infinite, any path, being a sequence $(s_0, s_1, \ldots)$ of states such that $(s_i, s_{i+1}) \in T$, can therefore only be of *finite* length. And finally, to define the labeling function L we use the natural valuation, that is, for each node $s \in S$ of the process we define a dedicated variable which is equal to true at that node only. For ease of reference, we name this variable the exact same as the node itself. Individual CTL^+ constraints are evaluated on the model $\mathcal{M}$ using state space enumeration. Then, a constraint ϕ is valid for the process P iff $\forall s \in S : \mathcal{M}, s \models \phi$, where $\mathcal{M}$ is the corresponding model of P (see Definition 3.3).

3.6.2 Validation Algorithm

Although many model checkers exist, we specified a simple search algorithm using state space enumeration to test the feasibility of model checking business processes. In doing so, we did tailored the algorithm for the specific use of business processes with finite paths. But, although the results are positive, we are certain that great advances in computation time can be achieved through the introduction of a more efficient algorithm.

The validation algorithm is implemented through a package containing classes with a one-to-one mapping of the CTL^+ symbols described in Appendix A. As a result, any correct CTL^+ formula is supported by the algorithm, enabling easy extensibility of the set of constraints described earlier. The core algorithm consists of

- StateQuantifiers;
  - All, Exists;
  - Implies, Proposition;

  - Or, And, Negation.

- PathQuantifiers;
  - Next, Finally, Globally, Until, Unless;
  - Or, And, Negation.

The StateQuantifiers *All* and *Exists* take a single PathQuantifier as argument. *Implies* takes two StateQuantifiers as arguments, and *Proposition* — which resembles an atomic formula — takes a node or node type as an argument. The PathQuantifiers *Next*, *Finally*, and *Globally* take a single StateQuantifier as an argument, whereas *Until* and *Unless* take two. The quantifiers *Or*, *And*, and *Negation* take only their own type as input. Through these specific interactions, a set of tree-like constructions representing only correct CTL^+ formulas can be formed. As an example, the CTL^+ formula $p \Rightarrow A[qUr]$ will be constructed in the following way: $Implies(Proposition(p)$ $, All(Until\ \ (Proposition(q),\ Proposition(r))))$.

Both StateQuantifiers and PathQuantifiers implement the *validate* method, which moves through a process tree employing the *validate* methods of the quantifiers which are nested into the current quantifier until a correctness decision is reached. We discuss the *validate* methods of the non-trivial core elements. To increase readability, these methods lack those lines and arguments that are used for the purpose of optimization and a number of safeguard checks, but remain the same in their essence.

Listing 3.1: Validate Method of the StateQuantifier All

```
public boolean validate(CTLNode e){
  Iterator<List<CTLNode>> paths =
    CTLUtil.getAllPathsFromNode(e).iterator();
  boolean ret = paths.hasNext();

  while(paths.hasNext() && ret)
    ret = q.validate(paths.next());

  return ret;
}
```

Listing 3.1 illustrates the *validate* method for the *All* StateQuantifier. The *validate* method takes a node e of the process tree as input. It then requests all paths from this node e and initializes its variables. In case there are no paths returned the method returns false. However, in practice the paths returned will always include the CTLNode e as its first element and therefore should never be empty. For each of those paths, the *validate* method of the nested PathQuantifier q is called until one

returns false. When all paths return positively, then the method returns true, and false value is returned otherwise.

Listing 3.2: Validate Method of the StateQuantifier Exists

```
public boolean validate(CTLNode e){
  Iterator<List<CTLNode>> paths =
    CTLUtil.getAllPathsFromNode(e).iterator();
  boolean ret = false;

  while(paths.hasNext() && !ret)
    ret = q.validate(paths.next());

  return ret;
}
```

Listing 3.2 illustrates the *validate* method for the *Exists* StateQuantifier. The *validate* method operates in the same manner as its counterpart of the *All* StateQuantifier, but returns true as soon as one path returns true.

Listing 3.3: Validate Method of the PathQuantifier Next

```
public boolean validate(List<CTLNode> path){
  return path.size() > 1 && p.validate(path.get(1));
}
```

Listing 3.3 illustrates the *validate* method for the *Next* PathQuantifier. The *validate* method receives a path as input, checks if a next element exists, calls the validate method of its nested quantifier p for that next element, and returns the result.

Listing 3.4: Validate Method of the PathQuantifier Finally

```
public boolean validate(List<CTLNode> path){
  boolean ret = false;
  Iterator<CTLNode> pathIt = path.iterator();

  while(pathIt.hasNext() && !ret)
    ret = p.validate(pathIt.next());

  return ret;
}
```

Listing 3.5: Validate Method of the PathQuantifier Globally

```
public boolean validate(List<CTLNode> path){
  Iterator<CTLNode> pathIt = path.iterator();
  CTLNode n = null;
  boolean ret = pathIt.hasNext();
```

```
  while(pathIt.hasNext() && ret){
    n = pathIt.next();
    if(!(n instanceof CTLLoopNode))
      ret = p.validate(n);
  }
  return ret;
}
```

Listing 3.4 illustrates the *validate* method for the *Finally* PathQuantifier. The *validate* method receives a path as input, and initializes its variables. It then loops through the path and calls the *validate* method of its nested quantifier p for each of the elements of the path until a positive result has returned. In this case the loop is interrupted and true is returned immediately. In case the end of the path is reached without a positive result then false is returned. The *validate* method for the *Globally* PathQuantifier is depicted in Listing 3.5. It operates in almost the same way as its counterpart of the *Finally* PathQuantifier, except that it requires a positive result along the entire path in order to return a true value. In cases where a loop is detected at the end of a path, p holds globally for the infinite loop and a true value is returned.

Listing 3.6 illustrates the *validate* method for the *Until* PathQuantifier. Only the *Until*'s PathQuantifier is discussed here as the *Unless*'s PathQuantifier is very similar. The *validate* method of the *Until* requires the quantifier p (which is the first of the nested quantifiers) to hold in the path until the moment when the second nested quantifier q holds. After initializing its variables, the *validate* method loops through the path and calls the *validate* methods of both p and q. While the validate method of p returns positively, it continues looping through the path. When the validate methods of both p and q do not return positively, the loop is interrupted and false is returned. In case the validate method of q returns positively, and the validate method of p has returned positively so far, the loop is interrupted and true is returned. In all other cases false is returned.

Listing 3.6: Validate Method of the PathQuantifier Until

```
public boolean validate(List<CTLNode> path){
  Iterator<CTLNode> pathIt = path.iterator();
  CTLNode n = null;
  boolean ret = false;
  boolean ok = pathIt.hasNext();

  while(pathIt.hasNext() && !ret && ok){
    n = pathIt.next();
    if(!(n instanceof CTLLoopNode)){
      ok = p.validate(n);
```

```
      ret = q.validate(n);
    }
  }
  return ret;
}
```

Finally, the proposition StateQuantifier is an abstract entity and therefore is not listed here. Instead, several child elements of this quantifier exist. The most common of these is the one which checks if a particular element in the process tree is actually the current node in the path. One of the others, for example, checks if the current node in the path is a start $\odot$ or an end $\otimes$ event. Using different combinations of the elements discussed here, any correct $CTL+$ formula can be represented and validated. And, as a result, the set of constraints and formulas discussed above can be easily extended due to the modular design of the validation algorithm.

3.7 Evaluation

To evaluate the proposed PVDI, we compare it with the more common imperative techniques on the grounds of their expressive power and the complexity of design. Given that there are no metrics or benchmarks available, we therefore begin by providing a framework for the evaluation. For defining the boundaries of the comparison, we give a definition of both the imperative variability and the declarative PVDI variability. We then define several basic properties which we use to identify expressive power features. And finally, we explore the difference in complexity, intended as the number of variants that can be described compactly with a given approach.

3.7.1 The Imperative Case

Imperative process specifications focus on a specific process definition by using transitions to prescribe the order of node traversal [Balko et al. 2009]. These structural variations adapt a process by applying a list of atomic operations in a specific order to the business process. Such operations for example include the replacement of an activity by another one, the addition of a flow, or the removal of a process fragment [Aiello et al. 2010, Weber et al. 2008]. Since different structural variations can contradict each other, it is necessary to specify which structural variations may or may not be applied together using *variation relations*. We call the combination of these two mechanisms the approach of *variation points*.

An example of a variation point is illustrated in Figure 3.11d. The upper branch contains two hexagonal tokens signifying a variation point where either activity “C” or activity “D” may be included.

Definition 3.24 (Imperative Template)**.** *An* **imperative template** *R is a tuple $\langle P, VP \rangle$ where P is a process and VP is a set of variation points.*

When one or more structural variations are selected from the template, the resulting process is called a *variant*. A variant may only contain structural variations as allowed by the variation relations between the different variation points in the template. A process containing combinations of structural variations which are not allowed by the variation relations is therefore not a variant. Imperative variability is the ability to produce a variant V by selecting a set of structural variations from a template.

3.7.2 The Declarative PVDI Case

Declarative process specifications define relationships between tasks in the form of constraints, allowing any paths as long as these constraints are not violated [Balko et al. 2009]. PVDI is based on this approach, but with one important difference. Instead of interpreting constraints on the state space of the process graph, PVDI evaluates them on the graph itself. We use the PVDI definitions for constraints, templates, and variants as provided in Section 3.2 to evaluate the declarative expressive power of PVDI and then to compare it with the imperative case. Declarative variability as used by PVDI is the ability to produce multiple non-bisimilar variants from a template R for which every constraint $\phi \in R_\Phi$ evaluates to *true* at every node $s_i \in v_S$ of every variant v. Non-bisimilar variants consist of those variants which do not effectively simulate each others behavior and thus offer unique process flows [Milner 1989].

3.7.3 Expressive Power

Templates are used in both variability techniques to capture a process plus the available variability. Given that we are interested in comparing of the two approaches mentioned above, we preliminary define the properties of a template to be finite and closed.

Definition 3.25 (Finite and Closed Templates)**.** *A template R is* finite *iff it has only finitely many variants. It is* **closed** *iff the set of nodes of every variant of R is contained in the set of nodes of R.*

The second property describes closed templates. A template is closed if and only if for all possible variants based on that template, the set of nodes is a subset of the set of nodes in the template. In other words, no new node can be introduced to

variants. This entails that any template which is closed is also finite, and thus offers only a limited and very specific set of variants as stated in the following theorem.

Theorem 3.1. *Any template R that is closed is also finite.*

Proof. A closed template R produces a set of variants V such that $\forall v \in V : v_S \subseteq R_S$. Since $|R_S| \in N$, disregarding constraints, only a limited number of transitions V_T can be drawn between the nodes in R_S. Meaning $|V_T| \in N$. It follows that $|V| \in N$, and from the definition that R is finite. □

Imperative variability is expressed through templates which include variation points. The expressive power of imperative variability is therefore directly connected to the template itself and the available set of atomic operations [Aiello et al. 2010, Weber et al. 2008]. Theorem 3.2 shows that imperative templates are both closed and finite. As a result, all variability offered through imperative templates must be specifically designed and prescribed within the template.

Theorem 3.2. *All imperative templates are closed and finite.*

Proof. An imperative template R consist of a set of nodes R_S, transitions R_T, and a set of variation points R_{VP}. An imperative variant V consists of an imperative template V_R and a subset of structural variations $SV \subseteq VP_{SV}$ chosen from those in the variation points of the template $V_{R_{VP}}$. Therefore, all nodes in variants $V_S \subseteq R_S$. From the definition it follows that R is closed, and from Theorem 3.1 it follows that R is finite. QED. □

Imperative variability frameworks however do sometimes include techniques which increase the expressive power of the framework, allowing for non-closed and non-finite templates. A common example is a so–called *placeholder node*. A placeholder is a place in a template which may or may not be used to include a new node. In PVDI terms it may be described as $p \Rightarrow (AXq \vee AXAXq)$, where the placeholder is preceded by p and followed by q. As a result, a template including such placeholders becomes not closed, nor finite, and therefore is more expressive. Other frameworks allow structural variations within structural variations, a powerful construct that can easily lead to inconsistent and unmanageable variants. Allowing this, does break the finite property of imperative templates, while the closed property does remain valid since all activities in the variant remain a subset of those included in the template.

Contrary to imperative variability, variability in PVDI is not defined explicitly within templates. Instead, the variability is offered through the underspecification of the process. Those variability options which are not allowed are specifically disallowed through constraints. The expressive power of PVDI is therefore directly

related to the ability to disallow one thing and to allow others. In other words, to disallow exactly enough without allowing unwanted possibilities. Because of the approach of underspecification we know from Theorem 3.3 that PVDI templates have the option of specifying templates in such a way that they are neither closed nor finite.

Theorem 3.3. *There exist PVDI templates that are neither closed nor finite.*

Proof. Consider a PVDI template R, which consists of a set of nodes $R_S = \{p, q\}$, a set of transitions $R_T = \emptyset$, and a set of constraints $R_\Phi = \{p \Rightarrow AFq\}$. Variant V based on this template consists of the template $V_R = R$ a set of nodes $V_S = \{p, q, r\}$, and a set of transitions $V_T = \{p \rightarrow r, r \rightarrow q\}$. All constraints R_Φ valuate to *true* at all nodes $s_i \in V_S$. Because V is a variant and $V_S \nsubseteq R_S$ we conclude that R is not closed. Since we may replace $r \in V_S$ with any node and produce a variant, we also conclude R is not finite. □

Declarative frameworks sometimes include imperative techniques. This does not increase their theoretical expressiveness, but rather helps to improve their expressiveness in practice. For example, in [Lu et al. 2009, Sadiq et al. 2005] the authors use an imperative process structure with pockets of declarative variability, and in [Groefsema et al. 2011] we propose to capture imperative operations with the help of the sets of constraints. We extended our proposal in Section 3.2.2.

3.7.4 Ease of Use

It is difficult to make a clear comparison between imperative and declarative approaches regarding practical use in terms of their complexity. This is mainly due to the fact that their usability varies depending on each particular case and the particular variability tool or framework in use. In practice, imperative approaches are useful when dealing with templates with limited flexibility. On the other hand, declarative templates offer a great deal of flexibility, which is useful in the case of highly variable business processes but turns out to give large overheads when dealing with templates with limited flexibility. Next, we define a framework for the comparison and we quantitatively compare the relative complexities of both approaches regarding their ease of use.

To give an impression of how this happens in practice consider the example in Figure 3.9 illustrating the issue arising when implementing a highly variable template utilizing imperative techniques. All the combinations shown in the figure must be implemented explicitly as nine variants, and each of those variants is made of variation points (shown as pairs of hexagons with dashed connector between them). In result, a process modeler should choose one of the nine predefined

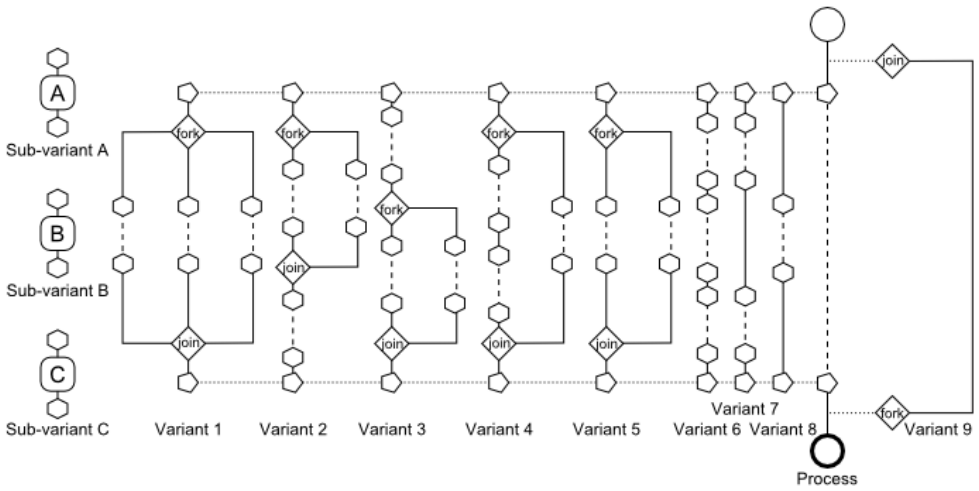


Figure 3.9: *High variable imperative solution.*

variants, and then choose how to fill the placeholders with a variant of choice. The template includes more than 50 possible variations in total, which must be provided at the template level either explicitly or by allowing to fill the placeholders by different activities. The difficulty of the task even increases in situations where some of the combinations are not allowed. Such restrictions should be reflected in the template, for example, by linking a list of possible options to particular placeholders. In the worst case, each of the more than 50 possible options should be visited at the stage of template modeling in order to decide if it is allowed or not. On the other hand, the same task can be easily solved using PVDI techniques. Only two formulas are required in order to make the specification: $start \Rightarrow AF(A \vee B \vee C)$ and $(A \vee B \vee C) \Rightarrow AFend$. The PVDI equivalent of this can be seen in Figure 3.10. In order to disallow some combinations, additional constraints can be added in order to reflect the rules which restrict the possible customizations.

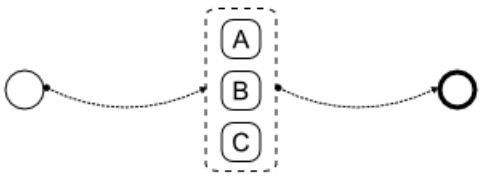
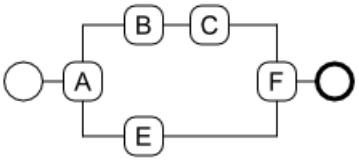
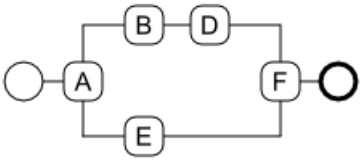


Figure 3.10: *High variable declarative solution.*

The situation changes when dealing with low–variable cases. Consider for instance two possible variants shown in Figures 3.11a and 3.11b. They only differ in



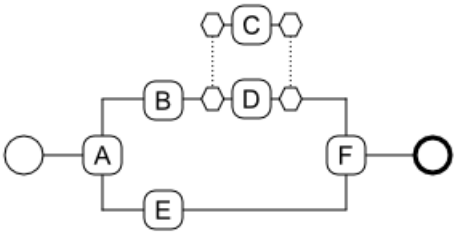
(a) Variant 1.



(b) Variant 2.

$start \Rightarrow AX\,a$
$a \Rightarrow EX\,b$
$a \Rightarrow EX\,e$
$a \Rightarrow AX\,(b \vee e)$
$b \Rightarrow AX\,(c \vee d)$
$c \Rightarrow AX\,f$
$d \Rightarrow AX\,f$
$e \Rightarrow AX\,f$
$f \Rightarrow AX\,end$

(c) Declarative Solution.



(d) Imperative Solution.

Figure 3.11: *Two variants (a) and (b) and their declarative (c) and imperative (d) solutions.*

one activity: either C or D is included in the upper branch following the activity B. An imperative template is simple, as illustrated in Figure 3.11d, but the things become different when one tries to convert this template into a set of formulas. The set of formulas in Figure 3.11c shows just one of the possible options which comprises nine formulas. Solutions will obviously become rapidly more complicated for larger processes. These low– and high–variable cases offer valuable insights into the complexity of both the imperative variability and the declarative variability offered by PVDI. In order to explore their complexity further, let us first define a task of variability management as a non-empty set of possible variants $V_A = \{v_0, \dots\}$, each being a process made of activities of some finite set of activities A. Every variant represents a single legal modification of some business process, which is typically referred to as template process. We can then define the complexity of a given variability approach regarding a variability task V_A as the number of different structural variations or the number of constraints needed to describe this task as a function of the cardinality of the set A.

While considering imperative variability approaches, the complexity is directly related to the amount of structural variations needed in a template in order to express all of the possible variants. Thus, each variant $v \in V_A$ is the result of applying of one or more structural variations. Therefore, the minimal number of structural variations is greater than or equal to the number of possible variants. Since the complexity equals the amount of structural variations, the complexity itself is also

greater than or equal to the number of possible variants. Therefore, the minimal theoretical complexity equals to 1 in the case of only one possible variant. The maximum theoretical complexity can be considered when there are no restrictions at all, that is, the set V_A contains all possible business processes which can be built based on the activities of the set A. To estimate that number, consider Theorem 3.4.

Theorem 3.4 (Theoretical complexity of imperative variability)**.** *The number of imperative structured variations for a business process build of N distinct activities and a finite number of gateways is at least $O(5^N)$.*

Proof. The maximal number of structured variations which do not involve the adding of removing of activities is equal to the number of non–equivalent business processes build of the same activities. This number we can use as a lower bound to estimate the number of possible structural variations.

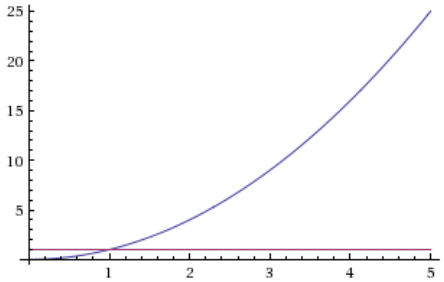
For a given business process, built of activities $a_1, a_2, \ldots a_N$, pick an arbitrary activity, for example, let it be a_1. Then, for each pair of activities a_1, a_k (where $2 \leq k \leq N$), the following options are possible:

1. Activity a_1 is always followed by activity a_k;
2. Activity a_1 is followed by activity a_k, but not always;
3. Activity a_k is always followed by activity a_1;
4. Activity a_k is followed by activity a_1, but not always;
5. Neither of the above statements is true, meaning that activities a_1 and a_k are parallel.

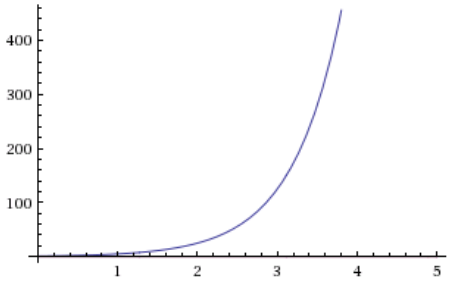
In total this results in $N-1$ pairs with 5 options per pair, and thus 5^{N-1} possible combinations. It must be noted that the number 5^{N-1} gives only the lower bound since we left out of consideration the possible relations between the other activities. As a result, the lower bound of the number of distinct business processes is of $O(5^N)$. □

Let us consider the declarative-based variability used by PVDI. In this case, the complexity equals to the number of constraints included in a template in order to restrict the number of possible variants. In the extreme case of high variability, when there are no restrictions at all, a PVDI template is quite simply a process. Which gives us a constant complexity of 0. The other extreme occurs when no changes are allowed. Because it captures the complete process structure instead of single transitions, we use the frozen area Definition 3.15. A frozen area requires $2N + 1$ formulas, each of length N, which gives in result $O(N^2)$ atomic formulas. Note that

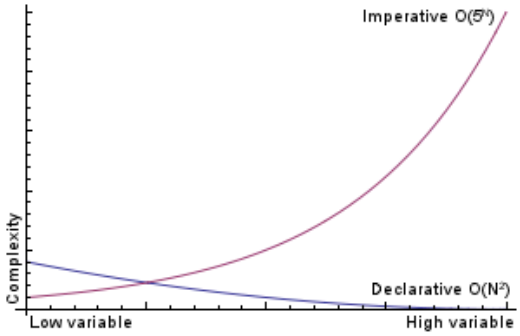
other low–variable cases can also be represented as a set of frozen or semi–frozen areas, which again results in a complexity of $O(N^2)$.



(a) Low variable case with constant (1) imperative and N^2 declarative complexity



(b) High variable case with constant (0) declarative and 5^N imperative complexity



(c) Both figures plotted against each other

Figure 3.12: *Imperative versus declarative complexity.*

To summarize, in our proposed framework of comparison between imperative and declarative approaches, the relative complexity is constant versus $O(N^2)$ for the case of a fixed process (Figure 3.12a) and $O(5^N)$ versus constant for the case of a highly flexible process (Figure 3.12b). The immediate conclusion is that employing the PVDI approach significantly reduces the upper bound of the complexity, because this approach always results in a polynomial complexity whereas imperative ones range widely from constant to exponential complexity. Figure 3.12c illustrates this fact by depicting an indication of how the complexity of a template increases on the average for both variability techniques when capturing a low to high amount of variability.

3.8 Implementation and Performance Results

PVDI tooling is supported with the development of VxBPMN (Figure 3.13). VxBPMN is developed using the Java programming language. It supports both the modeling of templates and variants using the Business Process Modeling Notation (BPMN) [Object Management Group (OMG) 2009] extended with the PVDI graphical elements (Figures 3.4-3.5). Business process models are saved in XML Process Definition Language (XPDL) format [van der Aalst 2003] which was extended in order to support PVDI flows and groups. The tool generates the CTL^+ formulas from the PVDI elements using the definitions provided earlier in this chapter. These formulas can then be used to model check the process model during and after its design.

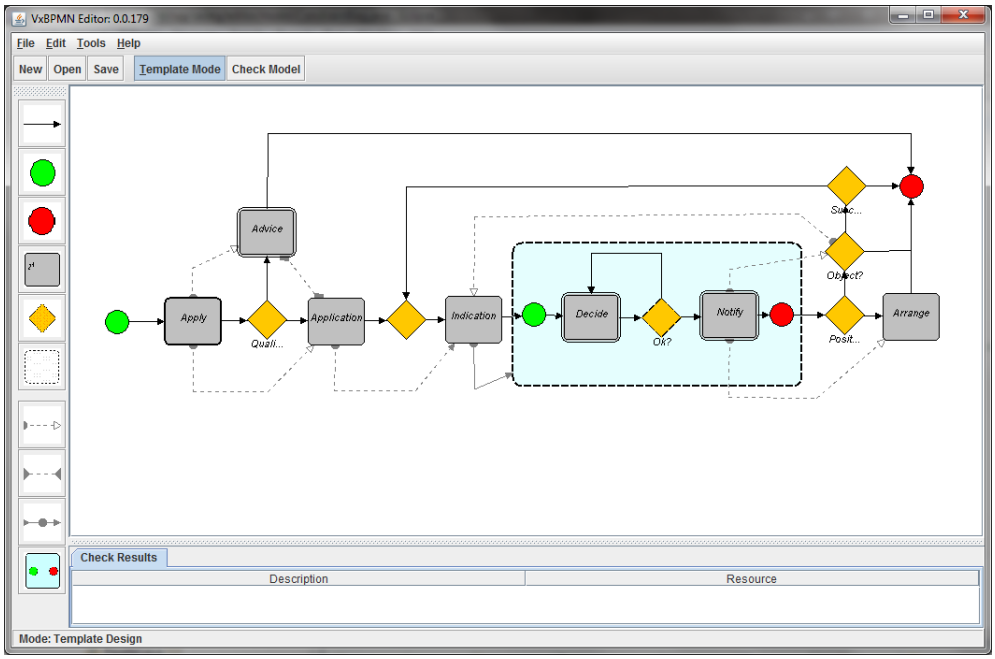


Figure 3.13: *VxBPMN PVDI tool.*

A performance test of the validation algorithm was conducted using a machine with an Intel® Core™ i7 950 at 3.07GHz, 6GB RAM (3×2GB Triple Channel), and an Intel SSD SA2M080G2GC running Windows 7 SP1 (64-bit) and Java 6 update 23 (64-bit). The performance test consisted of the evaluation of three different business processes consisting of 13 to 20 nodes, 15 to 21 transitions, and up to 1 frozen area. Each evaluation consists of the valuation of the business process embedded constraints, including soundness constraints, at every node. The number of con-

		Time (ms)		Model			Constraints	
#	Runs	Total	Avg	Nodes	Transitions	Frozen	Number	Violations
1	1000	3525	3.5	13	15	0	18	0
2	1000	1511	1.5	18	21	1	28	0
3	1000	599	0.6	20	20	1	26	2

Table 3.3: *Constraint valuation performance*

straints ranged from 18 to 28, where each frozen area counts as one constraint. Since frozen areas consist of a large number of constraints, the number of constraints for the two processes containing a frozen group was actually much higher. Table 3.3 contains the results of the constraint valuation performance test conducted with the VxBPMN tool. Every valuation was run one thousand times in order to get a fair average result and took between 599 and 3525 milliseconds for all thousand runs. On average, each individual process valuation took 3.5 milliseconds for the test No. 1, 1.5 milliseconds for the test No. 2, and 0.6 milliseconds for the test No. 3. Although on the first sight tests No. 2 and 3 seem quite similar, except for the two constraint violations, the difference of 1 millisecond can actually be explained by the difference in the complexity of the processes, which included a number of loops, and therefore included a much higher number of paths.

3.9 Discussion

Managing many variants of the same sort of processes is becoming a growing industrial need, especially to achieve mass customization and adaptation to several execution contexts. This has prompted for models and frameworks to deal with variability in an explicit manner in the field of business process management. Two techniques have emerged: imperative and declarative ones. Both are expressive and offer advantages for the trade of business processes engineering. Imperative techniques are well suited for the situations when variations are simple and there are not many of them. Also, it is usually to control the customizations when the imperative approach is employed. However, since possible variations must be modeled explicitly in advance, the situation becomes difficult when the number of variations grows.

Declarative approaches, on the other hand, excel in the case of a very flexible business process design, but usually face difficulties in expressing a straightforward enumeration of fixed variants. A natural conclusion is thus to choose the right approach which unites the best of the former two approaches. Though, this would require tool support for both techniques and possibly the necessity of migration from

one tool to another, with all the implied drawbacks. A better option is to combine both techniques while limiting the practical, but not theoretical complexity. As discussed above, imperative techniques have a complexity ranging between 1 for low variable templates to $O(5^N)$ for high variable templates. Declarative techniques, on the other, hand have a complexity ranging from $O(N^2)$ for low variable templates to 0 for high variable ones.

When comparing one approach with another, the one which has the lowest complexity is definitely more attractive. Unfortunately, the exact complexity of each of the two approaches seriously depends on the particular case under consideration. Our frameworks effectively solves this dilemma by employing a declarative approach together with the support for imperative methods. These imperative methods are internally encoded in as a set of declarative formulas, but from the user's point of view those formulas are hidden beneath visual modeling elements. By employing a graphical approach, PVDI allows a template designer to use a mix of both techniques while while staying away from the complexity issues of both techniques.

With PVDI we ameliorate a number of common BPM issues such as reusability and flexibility, as well as common BPM variability issues relating to complexity and usability. We do so by offering a graphical design extension to BPMN which internally transposes the designer's wishes into constraints which are then automatically processed by the model checker. Through the addition of the graphical layer the complexity of the declarative formalism is completely hidden from the business process modeler. Additional graphical notations describing new behavior can be added easily to the extensible PVDI framework, offering an even larger range of options than described in this chapter. The foundation of PVDI is rooted from a declarative base, which we proved to be the less complex choice.

In Chapter 2, some of the state of the art design–time frameworks were described, including Configurable Workflow Models [Gottschalk et al. 2008] and business process modeling with explicit variability support [Reichert and Dadam 1998, Sun and Aiello 2008]. Unlike those frameworks, PVDI offers both declarative and imperative variability techniques. Because of its declarative foundation, its expressive power is bound by the lower bound in complexity, whereas most other frameworks are bound by the higher imperative complexity. At the same time, declarative usability issues are ameliorated through a straightforward to use visual modeling extension from which the declarative constraint formulas can be generated directly.

Imperative business process variability techniques adopted in PVDI by the means of Frozen Area (Section 3.2.2) and its modifications allow PDVI to provide the possibilities similar to those provided by the state of the art annotation–based variability frameworks. At the same time, the usage of pure declarative features, such as ordering relations, allows to build a highly flexible business process definition, which

is provided by the constraint–based business process management tools. To conclude, PVDI covers both imperative and declarative branches of business process variability, thus allowing an end–user to benefit from the features of both sides.

The visual modeling paradigm of PVDI has some affinity with DECLARE and DecSerFlow frameworks, and in all cases the set of visual elements and their corresponding constraints is extendable. Nevertheless, in the case of PVDI the model is reaches because of the grouping elements which bring the traits of imperative variability which cannot be easily reached using the frameworks mentioned above.

Partly published as:
P. Bulanov and A. Lazovik and M. Aiello – *"Business Process Customization using Process Merging Techniques,"* Int. Conference on Service-Oriented Computing and Applications (SOCA–2011), pp. 1–4, 2011.

Chapter 4

Business Process Transformation

The issue of business process evolution arises when an organization is large enough to have many departments or affiliates which work in their specific ways. In result, there may be several business process specifications of the same task over different departments, where the variations in the specifications are driven by the local requirements of those particular departments. In such situation, the number of business process models grows because of the introduction of different variants, which makes the task of change management more difficult.

One of the typical ways to address the issue is to make a special business process, also known as a *template* or *reference* process, which holds the general outline of the process along with some overall recommendations. Then, such template can be used either as it is or it can be *customized* in order to fulfil the specific requirements of a particular department. However, this approach does not help when one needs to make some global amendments to be applied across the whole organization.

As shown in Figure 4.1, the arrows "1" and "2" reflect the fact that the process on the right side (or the bottom side, accordingly) of the arrow is made by a customization of the template process. If such a customization can be represented as a transformation function, then it can be repeated automatically without human effort. In Figure 4.1, function f represents the transformation from the template process into one of its variants, and function g represents the transformation from the template process into the new template. The arrows "3" and "4" represent the application of combinations of the functions f and g in different order.

If the transformations are represented in some formal way, then they may be applied automatically for each of the possible variants. A process modeler who is going to modify the template process could immediately modify all of the variants or receive a feedback if such a propagation is not possible, including the list of variants which failed to be transformed.

In this chapter, we introduce a novel language to facilitate a representation of a business process as a set of formal logic formulas. Then, we analyze how different

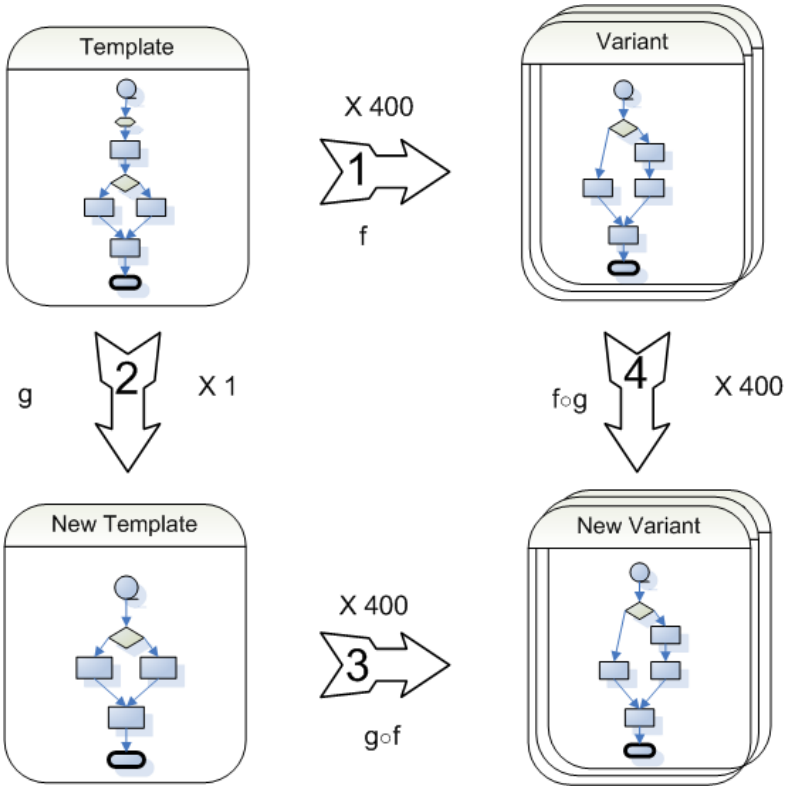


Figure 4.1: *Business Process Evolution*

manipulations with business process model can be reflected in its formal representation, which in turns gives a key to represent a modification of a business process model as a function which transforms a set of formulas into another set of formulas. And finally, we show how such function can be extracted from two business process models – original and target ones, and provide performance evaluation on a test set of realistic business processes.

4.1 A Temporal Logic of Business Processes

Formal languages have often been the foundations for BPM automation. Much researched examples include Petri-Nets [Petri 1962, Murata 1989], Process calculi [Milner 1999] and Temporal Logic[1]. Our approach uses the latter as the basis for

[1]See Appendix A

defining a language suitable to represent a business process. We achieve this via taking into account the specifics of business process models, namely, the presence of different types of branching points (which are usually called gateways). We begin by defining the formal language for processes, then we provide a semantics for the language, identify basic properties and move onto its application for business process representation and transformation.

4.1.1 Temporal Process Logic

The Temporal Process Logic (TPL)[2] is a modal propositional language that can talk about the truth of propositions in future states, but also unlike LTL and CTL languages can take into account the differences between possible execution runs. It was initially introduced in [Bulanov, Lazovik and Aiello 2011], and herein we reuse it for the case of business process transformations. The underlying processes are considered to have AND and OR gates in addition to simple branching from one state to another one. Its syntax is quite straightforward: we have a set of proposition symbols AP, propositional unary and binary operators $\neg, \wedge, \vee$, plus three unary modal operators $\rightarrow, \rightsquigarrow, \rightrightarrows$. The intuitive meaning of the last operators is the following.

$\rightsquigarrow a$ means there is a state satisfying a in the process, and this state can be reached from the current state following the process model.

$\rightarrow a$ means there is a state satisfying a in the process and that this state can always be reached from the current state following the process model. In case of parallel splitting with an AND-gate, at least one of the parallel branches must lead to the state satisfying a.

$\rightrightarrows a$ is the same as $\rightarrow a$, but in case of any parallel splitting, all of the parallel branches must lead to a state satisfying a.

The following statements are true of TPL formulas for any proposition a:

$$\rightarrow a \Rightarrow \rightsquigarrow a \qquad \rightrightarrows a \Rightarrow \rightarrow a$$
$$\rightarrow (\rightarrow a) \Rightarrow \rightarrow a \qquad \rightrightarrows (\rightrightarrows a) \Rightarrow \rightrightarrows a$$
$$\rightarrow (\rightsquigarrow a) \Rightarrow \rightsquigarrow a \qquad \rightrightarrows (\rightarrow a) \Rightarrow \rightarrow a$$
$$\rightsquigarrow (\rightsquigarrow a) \Rightarrow \rightsquigarrow a \qquad \rightrightarrows (\rightsquigarrow a) \Rightarrow \rightsquigarrow a$$

[2]The term Temporal Process Logic has been used before in the context of concurrency [Cleaveland 1999]. Here by TPL we define a new language which is unrelated to the previous work that goes by the same name.

Process runs as TPL semantics

The idea is to evaluate TPL formulas over processes and to use them to describe the salient behaviors of the processes. We begin by providing a formal definition of a process with AND and OR gates. This definition actually reflects a formalization of a business process specified in terms of BPMN–like notation [Object Management Group (OMG) 2009], and it is in line with the definition of a process from Chapter 3.

Definition 4.1 (Process). *A* **process** P *is a tuple* $\langle A, G, T\rangle$ *where:*

- A *is a finite set of activities, including start activity* $\odot$ *and final activity* $\otimes$*;*
- G *is a finite set of gateways, each of type* $\{AND, OR\}$*;*
- $S = A \cup G$ *is a set of states;*
- $T = T_a \cup T_g$*, where:*
- $T_a : (A \backslash \{\otimes\}) \rightarrow S$ *is a function which assign a next state for each activity;*
- $T_g : G \rightarrow 2^S$ *is a function which assign a nonempty set of next states for each gateway;*
- *The graph* $G = \langle S, T\rangle$ *contains no cycles.*

When talking about a process P we refer to its set of activities as P_A, its set of gateways as P_G, and its set of transitions as P_T.

When a business process is executed by some process engine, at each point a decision has to be made which activity should be executed next. Such a choice is trivial for a linear business process, but becomes more complicated in the case of "branched" process. Formally, such a choice can be represented as a **choice function**.

Definition 4.2 (Choice function). *For a given business process* $P = \langle A, G, T\rangle$*, a* **choice function** $D : S \rightarrow 2^S$ *is defined as follows:*

- $D(\otimes) = \emptyset$*;*
- $\forall e \in S \backslash \{\otimes\}$*:*
  - *If* $e \in A$*, then* $D(e) = \{T(e)\}$*;*
  - *If* $e \in G$*, then, depending on the type of the gateway* e*:*
    - *AND:* $D(e) = T(e)$*;*
    - *OR:* $D(e) \subseteq T(e)$*,* $D(e) \neq \emptyset$*.*

For a given business process there can be one or more different choice functions. Given a choice function D, for each state a we can formally define the set of *reachable* states **reach**(a, D) as a smallest set $R \subseteq S$, such that $a \in R$ and $\forall x \in R : D(x) \subseteq R$.

A choice function represents a set choices made by an execution engine during invocation of a business process. With such set of choices it is possible to build its corresponding execution graph:

Definition 4.3 (Execution)**.** *An* **execution** σ *for a process* P *w.r.t. a choice function* D *is a directed graph* $\langle \widetilde{A}, \widetilde{T} \rangle$*, where*

- $\widetilde{A} =$**reach**$(\odot, D)$*, and*
- $\widetilde{T} \subseteq \widetilde{A} \times \widetilde{A}$*, built such as* $(a, b) \in \widetilde{T} \Leftrightarrow b \in D(a)$*.*

The choice function D *is called a* **basis** *of the execution* σ*.*

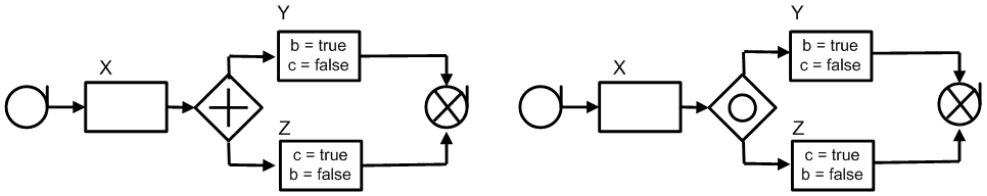


Figure 4.2: *Two process models*

In other words, an execution represents a single execution of an abstract business process engine such that for each OR–gate a different path is taken while for each AND–gate all subsequent sub–paths are included. The set of all possible executions of a process P is denoted as Ω_P or just Ω.

In Figure 4.2, we present examples of two simple processes, each of them contains a start activity, an end activity, and activities X, Y, and Z. The only difference between them is in the gateway type. Then, Figure 4.3 shows all the execution paths for these processes. There is only one execution for the left process with the AND–gate, and there are three different executions for the other process.

We say that an activity a is followed by an activity b w.r.t. the execution σ and denote that as $a <_\sigma b$ when $b \in reach(a, \sigma_D)$, where σ_D is a basis of the execution σ. In other words, it means that there is a route in the graph σ leading from a to b. We say that an activity a is included into an execution σ and denote it as $a \in \sigma$ if $\odot <_\sigma a$.

We say that an activity a is strongly followed by an activity b w.r.t. execution σ and denote it as $a \ll_\sigma b$ when all possible routes in the graph σ which start in the activity a lead to b. Formally, the relation $\ll_\sigma$ is defined as smallest subset of $A \times A$, such that:

- $(a, a) \in \ll_\sigma \forall a \in \sigma$;
- $\forall x, z \in \sigma : x \neq \otimes \wedge (\forall y : y \in \sigma_D(x) \Rightarrow (y, z) \in \ll_\sigma) \Rightarrow (x, z) \in \ll_\sigma$.

In Figure 4.3 we have that $X < Y, X < Z$ for the left process, but it is not true that $X \ll Y$ (since there is a route which leads from X to Z and $\otimes$ and does not contain Y). As for the execution in the upper right part of the figure, both $X < Y$ and $X \ll Y$ are true; and the activity Z is not included into that execution.

Note that according to the definition of the process, each activity apart from the final one has the next step. Therefore, $\forall \sigma \in \Omega \; \forall x \in P_A \backslash \{\otimes\} : x \in \sigma \Rightarrow \exists y \in P_A : x <_\sigma y$.

Simplified TPL

Consider a business process P, and any two activities a and b of that process. With the help of the formalisms of the previous sections, it is possible to describe relative ordering of those two activities in the following way:

$$\begin{array}{l} a \rightsquigarrow b \Leftrightarrow \exists \sigma \in \Omega_P : (a \in \sigma \wedge b \in \sigma \wedge a <_\sigma b) \\ a \rightarrow b \Leftrightarrow \forall \sigma \in \Omega_P : a \in \sigma \Rightarrow (b \in \sigma \wedge a <_\sigma b) \\ a \rightrightarrows b \Leftrightarrow \forall \sigma \in \Omega_P : a \in \sigma \Rightarrow (b \in \sigma \wedge a \ll_\sigma b) \end{array} \tag{4.1}$$

TPL Truth definition

We can now establish the link between TPL formulas and the process models. Let AP be a set of propositional variables, and a labeling function $\nu : P_A \rightarrow 2^{AP}$ which assigns each variable in AP with a set of activities where that variable holds true. Now we can introduce the model $\mathcal{M} = \langle P, \nu \rangle$, where P is a process and ν is a valuation function. Now we can define the truth of a TPL formula in a model, note that the truth is local to activities of the process.

$$\begin{array}{l} \mathcal{M}, x \models a \Leftrightarrow x \in \nu(a) \\ \mathcal{M}, x \models \neg a \Leftrightarrow \mathcal{M}, x \text{ not } \models a \\ \mathcal{M}, x \models a \vee b \Leftrightarrow \mathcal{M}, x \models a \text{ or } \mathcal{M}, x \models b \\ \mathcal{M}, x \models \rightsquigarrow b \Leftrightarrow \exists \sigma \in \Omega_P : x \in \sigma \wedge \exists y \in \sigma : x <_\sigma y \wedge \mathcal{M}, y \models b \\ \mathcal{M}, x \models \rightarrow b \Leftrightarrow \forall \sigma \in \Omega_P : x \in \sigma \Rightarrow \exists y \in \sigma : x <_\sigma y \wedge \mathcal{M}, y \models b \\ \mathcal{M}, x \models \rightrightarrows b \Leftrightarrow \forall \sigma \in \Omega_P : x \in \sigma \Rightarrow \exists y \in \sigma : x \ll_\sigma y \wedge \mathcal{M}, y \models b \end{array} \tag{4.2}$$

The temporal unary TPL operators can be lifted to binary ones in the following way.

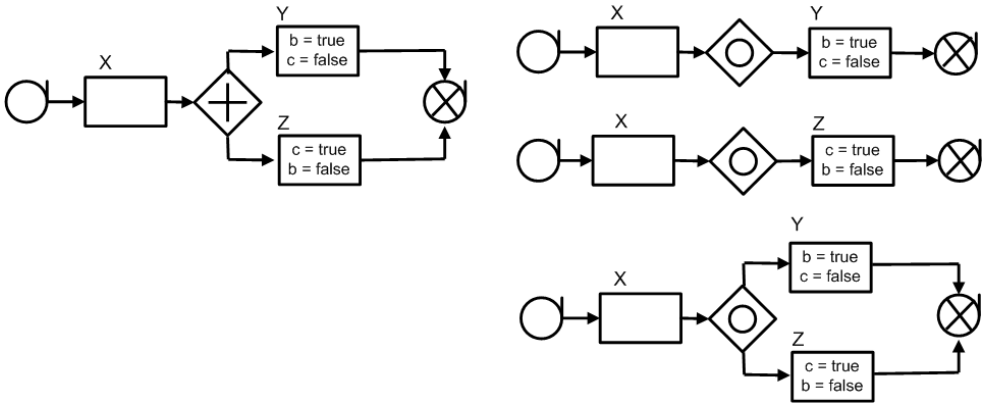


Figure 4.3: *Possible process executions*

$$\begin{aligned}
&\mathcal{M}, x \models a \rightarrow b \Leftrightarrow (\mathcal{M}, x \models a \Rightarrow \mathcal{M}, x \models\!\!\rightarrow b) \\
&\mathcal{M}, x \models a \rightsquigarrow b \Leftrightarrow (M, x \models a \Rightarrow \mathcal{M}, x \models\!\!\rightsquigarrow b) \\
&\mathcal{M}, x \models a \rightrightarrows b \Leftrightarrow (\mathcal{M}, x \models a \Rightarrow \mathcal{M}, x \models\!\!\rightrightarrows b)
\end{aligned} \tag{4.3}$$

4.1.2 Discussion

We have provided a language and a definition of its models, we have also given the truth definition. The natural next step is to consider the completeness of TPL with respect to the process models and identify the axiomatics and model properties. We consider this an important step that is beyond though the current treatment and we leave it for future investigation.

Having a more practical research agenda, we decide to focus on its practical use for variability management. To do so, we begin by exploring its relation to other fundamental TL logics such as LTL and CTL. Let us compare them on a simple example. Figure 4.2 shows two process models, which differ only for the gateway type.

If we ignore the gates and consider them as directed labeled graphs, they are indistinguishable from the CTL point of view (i.e., bisimilar [Milner 1989]). For instance, in the state X the formula EFb is true and AFb is false for both models. On the contrary, the TPL formula $\rightarrow b$ evaluated in X is true for the left process but false for the right one, and the formula $\rightsquigarrow b$ is true for both of them.

The language introduced in this chapter can be used to describe the temporal relations in a given business process, in a similar way with Temporal Point Algebra [Vilain and Kautz 1986]. However, the ability to explicitly specify the difference between AND– and OR– gateways gives the ability to describe a business process

more precisely. As it is shown later in this chapter, we can utilize TPL to abstract from a concrete graphical representation of a business process and consider a business process as a set of temporal formulas. Because of the nature of TPL logic, such a representation is more precise than the one which is based on Temporal Point Algebra or other types of temporal logics (LTL or CTL).

4.2 Process representation and transformation

Given a process P with a set of activities P_A, with $N = |P_A|$, consider $N \times N$ matrix, where each row i corresponds an activity $a_i \in P_A$, and each column j corresponds an activity $a_j \in P_A$. In each cell of the matrix we put a symbol of the alphabet $\mathcal{A} = \{\rightarrow, \rightsquigarrow, \bullet\}$.

In a cell (i, j) we put the following symbol:

- $\rightarrow$ if the formula $a_i \rightarrow a_j$ is valid;
- $\rightsquigarrow$ if the formula $a_i \rightsquigarrow a_j$ is valid, but the formula $a_i \rightarrow a_j$ is not;
- $\bullet$ if the formula $a_i \rightsquigarrow a_j$ is not valid;
- $\bullet$ for the cells on the main diagonal, i.e., if $i = j$.

Such a matrix of size $N \times N$ is called a **process matrix** of a given process. A process matrix built for a process P is denoted as $S(P)$.

Clearly, if a cell a_{ij} contains one of the arrows ($\rightarrow$ or $\rightsquigarrow$), then the symmetric cell must contain $\bullet$, since the process graph is supposed to be cycle–free. More formally,

Property 4.1 (Antisymmetry)**.** $\forall i, j : a_{ij} \in \{\rightarrow, \rightsquigarrow\} \Rightarrow a_{ji} = \bullet$.

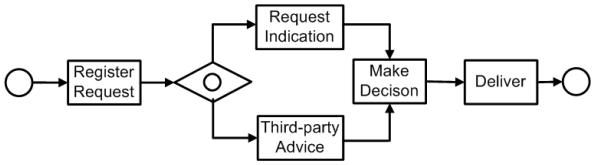
Also, due to the transitivity properties of TPL (described in section 4.1.1), a process matrix would inherit the transitivity traits:

Property 4.2 (Transitivity)**.** $\forall i, j, k$*:*

- $a_{ik} \in \{\rightarrow, \rightsquigarrow\} \wedge a_{kj} \in \{\rightarrow, \rightsquigarrow\} \Rightarrow a_{ij} \in \{\rightarrow, \rightsquigarrow\}$;
- $a_{ik} = \rightarrow \wedge a_{kj} = \rightarrow \Rightarrow a_{ij} = \rightarrow$.

Consider an example of a business process and its process matrix:

Example 4.1. *An example of a business process is represented in Figure 4.4a, and its process matrix is represented in Figure 4.4b. Here acronyms are used instead of activity names, ex., RR instead of Request Registration," RI instead of "Request Indication," and so on. Note that the second and third rows of the matrix (which correspond to RI and TA) are*



(a)

	RR	RI	TA	MD	DE
RR	•	$\rightsquigarrow$	$\rightsquigarrow$	$\rightarrow$	$\rightarrow$
RI	•	•	•	$\rightarrow$	$\rightarrow$
TA	•	•	•	$\rightarrow$	$\rightarrow$
MD	•	•	•	•	$\rightarrow$
DE	•	•	•	•	•

(b)

Figure 4.4: *Example 4.1*

identical, and the same is true for the second and third columns. And it is logical because those activities hold equivalent positions in the process, and if one exchanges them then the process itself would remain the same.

A process matrix for a given business process holds the Properties 4.1 and 4.2, the reverse situation is also true, as stated in the following theorem.

Theorem 4.1. *The matrix M over the alphabet $\mathcal{A}$ identifies a process iff the following properties hold:*

1. $a_{ii} = \bullet \forall i$;
2. *Property 4.1 (antisymmetry); and*
3. *Property 4.2 (transitivity);*

Proof. $\Leftarrow$ trivial, followed from the properties of process matrix
$\Rightarrow$ The proof is constructive by providing the description of the algorithm, which is schematically displayed in Algorithm 1.

This algorithm takes as an input the process matrix M and the set of activities A. It is presumed that the size N of the matrix M is equal to the size of the list A, and the order of activities in the list A is in accordance with the rows and columns in the matrix M.

At the initial step, the function *TransitiveReduction* is invoked in order to get rid of the redundant dependencies in the matrix M. A dependency is redundant iff it can be unambiguously restored using another dependency and the rules of transitivity.

A redundant dependency is removed from the matrix M (that is, the corresponding cells are cleared).

The algorithm uses a temporary working set Ω, which contains all activities which were already considered. Initially, this set contains only the final "activity" of the process - $\otimes$.

Then the main cycle starts. First, the function *GetPredecessors* is invoked. It searches for all such activities which refer to the ones which are already considered (which are contained in the set Ω) but do not refer to any other activities. Then, all those activities are marked as considered by adding them into the set Ω, and one iteration of the process structure is built by invoking the function *CreateNextElement*, which returns for each activity the next process element it must refer to (this elements can be either an activity or a gate).

The cycle is repeated as long as the function *GetPredecessors* returns non–empty set. Since the set A is finite, then the algorithm will finish the execution in finite number of steps.

The function *CreateNextElement* is described in details in Algorithm 2. Initially, for the given activity a, two sets of relations are built: one of type $\rightarrow$ and one of type $\rightsquigarrow$, respectively. The links are determined by reading the appropriate row in the process matrix. Then, if there are no relations of type $\rightsquigarrow$, a single AND–gate is built. Also, if the set M_A contains only one activity, then no gateway is built and this particular activity is returned. Then, for each activity a_i such as $a \rightarrow a_i$, the following check is made: if $\forall b_i : a \rightsquigarrow b_i : b_i \rightarrow a_i$, then the activity a will always be followed by the activity a_i, and there is no need to make additional link from a to a_i. Otherwise, the activity a_i is appended to the set S, which will be used later. If it happens that the set S is empty, then only an OR–gate is needed (line 15), otherwise, a combination of AND– and OR– gates is needed (line 18). If the set M_O contains only one activity, then a so–called bypassing OR–gate is created, with two outgoing link, one leading to the activity $b \in M_O$, and another leading to the process element immediately after the activity b.

Let us prove that at each step of the algorithm, the process being built corresponds the original process matrix. For a given activity a_i, if the process matrix contains the formula $a_i \rightarrow a_j$, then in result there will be either AND–gate leading to the activity a_j (Algorithm 2, line 5), or all the branches of an OR–gate will eventually lead to a_j (lines 15, 18). If the process matrix contains the formula $a_i \rightsquigarrow a_j$, then there will be an OR–gate, and only one of its branches would lead to a_j. That means, the formula $a_i \rightarrow a_j$ will not be valid for such process structure. If the process matrix contains $\bullet$ symbol in the cell M_{ij}, then the link from a_i to a_j will not be built while considering the activity a_i. Moreover, such a link cannot be added indirectly via another activity a_k, via adding links (a_i, a_k) and (a_k, a_j), since because

of transitivity property of the process matrix the cell M_{ij} must contain either $\rightarrow$ or $\rightsquigarrow$ element.

□

Algorithm 1 *Building a business process.*

INPUT: Process matrix M, list of activities A
OUTPUT: Process $\langle A, G, T \rangle$
TransitiveReduction(M)
$\Omega := \{\otimes\}$
repeat
    $\Phi :=$ *GetPredecessors*(P, Ω)
    $\Omega := \Omega \bigcup \Phi$
    for $a \in \Phi$ **do**
        $g :=$ *CreateNextElement*(a, M)
        $T := T \bigcup \{(a, g)\}$
    end for
until $\Phi = \emptyset$ **return** $\langle A, G, T \rangle$

As a consequence, any matrix which satisfies the Conditions 1–3 of the Theorem 4.1 actually identifies some business process, and this process can be built using the constructive proof of the Theorem 4.1.

Example 4.2. *Consider an example of building a business process basing on a given process matrix. The matrix is taken from the Example 4.1 and is repeated for convenience in Figure 4.5a.*

	RR	RI	TA	MD	DE
RR	•	$\rightsquigarrow$	$\rightsquigarrow$	$\rightarrow$	$\rightarrow$
RI	•	•	•	$\rightarrow$	$\rightarrow$
TA	•	•	•	$\rightarrow$	$\rightarrow$
MD	•	•	•	•	$\rightarrow$
DE	•	•	•	•	•

(a)

	RR	RI	TA	MD	DE
RR	•	$\rightsquigarrow$	$\rightsquigarrow$	—	—
RI	•	•	•	$\rightarrow$	—
TA	•	•	•	$\rightarrow$	—
MD	•	•	•	•	$\rightarrow$
DE	•	•	•	•	•

(b)

Figure 4.5: *Example 4.2*

First step is to run transitive reduction in order to remove the redundant dependencies, the matrix in Figure 4.5b shows the result of such reduction. The symbols "—" represent

Algorithm 2 *Function CreateNextElement.*

1: **INPUT:** Activity a, process matrix M
2: **OUTPUT:** Process element the activity a must refer to
3: $M_O := GetOrLinks(a, M)$ ▷ get the relations of $\rightsquigarrow$ type
4: $M_A := GetAndLinks(a, M)$ ▷ get the relations of $\rightarrow$ type
5: **if** $M_O = \emptyset$ **then return** *MakeGateway_AND*(M_A)
6: **end if**
7: $S := \emptyset$
8: **for** $a_i \in M_A$ **do**
9: **for** $b_j \in M_O$ **do**
10: **if** $\neg(b_j \rightarrow a_i)$ **then**
11: $S := S \bigcup \{a_i\}$
12: **end if**
13: **end for**
14: **end for**
15: **if** $S = \emptyset$ **then return** *MakeGateway_OR*(M_O)
16: **else**
17: $G :=$ *MakeGateway_OR*(M_O)
18: **return** *MakeGateway_AND*$(S \bigcup \{G\})$
19: **end if**

the places where appropriate formulas were removed. This matrix will be referred later as a working matrix.

Now we can run the main cycle of the Algorithm 1. In Figure 4.6, each line corresponds to a single iteration of the main cycle, where the second column contains the content of the set Φ *(which contains the result of invocation of function* GetPredecessors*). The third column contains the result of invocation of function* MakeLinksAndGateways*, which is actually the current state of the process building.*

Initially, the process to be build contains only the final step $\otimes$*. Then, the first call to function* GetPredecessors *returns one activity – DE (line 1 in the table), because this activity has no successors according to the working matrix. Function* MakeLinksAndGateways *would add this single activity, and its link to the final one.*

The second iteration is similar – function GetPredecessors *returns the activity MD, and* MakeLinksAndGateways *add it as well as its link to the next one — DE.*

At the third iteration, function GetPredecessors *returns the set of two activities –* $\{RI, TA\}$*. Next, they are both added into the process, as well as the links to their successors. In this case, the successor is the same – activity MD.*

At the fourth iteration, function GetPredecessors *returns one activity – RR. But while adding it into the process it turns that it has two successors – RI and TA. Therefore, a gateway must be introduced, and the type of this gateway is driven by the type of the dependency. According to the working matrix, the dependency type is* $\rightsquigarrow$*, and the gateway is therefore of type "OR."*

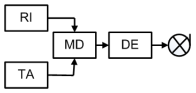
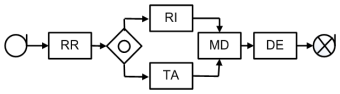
1		$\{DE\}$
2		$\{MD\}$
3		$\{TA, RI\}$
4		$\{RR\}$

Figure 4.6: *Example 4.2: Steps of the algorithm*

4.2.1 Transformations

Consider different kinds of atomic process manipulations and their matrix equivalence. The list of atomic operations was introduced in [Weber et al. 2008], and analyzed in [Bulanov et al. 2011], and herein we analyze several of those operations and how they can be represented as manipulations with process matrices.

1. Removing an activity from a process entails the removal of appropriate row and column from the process matrix, thus obtaining a process matrix of smaller size;

2. Adding a new activity entails the adding of a new column and row to the process matrix, and the elements in that row and column would identify the position of the newly added activity in the process;

3. Swapping two activities a_i and a_j entails the swapping of i^{th} and j^{th} rows and columns, while preserving the mapping between row/column numbers and the set of activities;

4. Move an activity a_i, such as it would be located parallel to an activity a_j entails the making the i^{th} row to be a copy of j^{th} row, and the same for the i^{th} column (while preserving the mapping between row/column numbers and the set of activities as well);

The manipulations above are called *primitive* process manipulations, and they are process-independent in a way that the same manipulation (ex., remove the activity "X") can be applied to any process, more formally:

Theorem 4.2. *Given a process matrix M and any primitive manipulation $pm(\cdot)$, then $M' = pm(M)$ is also a process matrix.*

Proof. We need to prove that the properties 1–3 of the Theorem 4.1 are retained after application of a primitive transformation.

For property 1, the proof is trivial.

For property 2 (anti–symmetry), consider each transformation individually

1, 2 Removing and adding rows and columns does not violate the anti–symmetry, provided that during adding of a new row and column it is filled with symbols $\bullet$.

3, 4 Swapping rows and columns does not affect the cells other than those belonging to i^{th} and j^{th} rows and columns.

For property 3 (transitivity), again consider each transformation individually

1 $\forall i, j, k$: if a_{ik}, a_{kj}, and a_{ik} are binded with transitive relation, then either i^{th}, j^{th}, or k^{th} column is removed then the transitive relation becomes irrelevant.

2 $\forall i, j$: consider the case the k^{th} row and column are added, and analyze how will it affect the possible transitivity relations between the elements a_{ik}, a_{kj}, and a_{ij}. Since the newly added row and column contain only $\bullet$ elements, then both a_{ik} and a_{kj} are equal to $\bullet$, therefore, no transitivity violation occurs. Now consider the elements a_{ij}, a_{jk}, and a_{ik}. Since both a_{jk} and a_{ik} are equal to $\bullet$, there there is no transitivity violation as well. The same is for the case of a_{ki}, a_{ij}, and a_{kj}.

3 Consider three matrix cells, a_{ik}, a_{kn}, and a_{in}, where index i is equal to the corresponding index in the permutation (that is, i^{th} row is swapped with j^{th}, and the same for columns), and k and n are arbitrary indices. The values of those three cells are binded with the transitivity property, now let us show that the permutation does not brake this link. Once the permutation is applied, the first two cells will take the values which were contained in the cells a_{jk} and a_{kn} respectively, and the third cell will hold the value which were in the cell a_{jn}. But the cells a_{jk}, a_{kn}, and a_{jn} were binded with the transitivity dependency in the original matrix, therefore, this transitivity property will also hold in the result matrix.

□

The primitive transformations 3 and 4 are actually permutations over the rows and columns of a process matrix. A permutation is basically (re)arranging of some list of objects. In our case, the objects to be rearranged are rows and columns of a process matrix, and such rearrangement is first applied to the rows of the input matrix, then the same rearrangement is applied to the columns of the intermediate matrix[3].

Such a permutation over a matrix of size N can be represented as a vector $V = \{n_1, n_2, \dots n_N\}$ of size N, each of its elements n_i being a natural number ranging from 1 to N. The i^{th} element of such a vector contains the target position of i^{th} row and column. A permutation $V = \{n_1, n_2, \dots n_N\}$ has no repetitions, if $\forall i, j = 1 \dots N : i \neq j \Rightarrow n_i \neq n_j$. In other words, all of the elements of the vector V are distinct. In short form, given a matrix A and a permutation vector V, the result of application of the permutation is written as $B = A^V$, where B is the result matrix.

In addition to primitive transformation, non–primitive ones can be introduced. They represent the direct manipulation with the content of the cells in a matrix in order to make arbitrary modifications. An example of such a modification is changing of the type of a gateway. Unlike primitive ones, there is nothing can be said about the applicability of non–primitive transformation on different process matrices in general, therefore, such transformations are process–dependent.

In order to make a formal representation of non–primitive transformations, consider a matrix Δ with elements belonging to the alphabet $\mathcal{A}' = \{\rightarrow, \rightsquigarrow, \bullet, -\}$, and specify the operation $\oplus$ such as:

$$\forall a, b \in \mathcal{A}' : a \oplus b = \begin{cases} a & \textbf{if } b = - \\ b & \textbf{otherwise} \end{cases}$$

The $\oplus$ operation can be expanded for matrices, therefore, a non–primitive transformation can be represented in matrix form as $B = A \oplus \Delta$, where Δ is a matrix over the alphabet $\mathcal{A}'$, and A is a process matrix. Note that the matrix B is not necessary a process matrix (that is, not necessary holds properties 2–3 of the Theorem 4.1).

In general, given two process matrices A and B, we can extract a transformation from A to B as a combination of primitive and non–primitive ones: $B = A^V \oplus \Delta$, where V and Δ represent primitive and non–primitive parts of the transformation, respectively. Note that such a transformation can always be extracted, since the matrix Δ can be chosen to be equal to the matrix B, but in such case the intended modification would be lost, and it will not be possible to reuse such a transformation

[3]Actually, the order of rearrangement can be reversed: first rearrange columns, then rows, with the same final result.

(since the result would be the same regardless the input). It is therefore beneficial to select from the family of possible transformations the one with minimal or possibly empty non–primitive part. The primitive part is considered to be empty if its matrix contains only elements "—."

In result, the task of finding a process transformation can be formulated in the following way: find such a permutation of rows and columns V in a matrix A, such that the difference between the transformed matrix A^V and the matrix B is minimal. The distance between matrices is measured as a number of non–equivalent cells.

Since the processes may be based on different sets of activities, then the preparation step is needed. Given two processes, P_1 and P_2, let us assume that the function which transforms P_1 into P_2 is sought. In that case, we add into the process P_1 activities which belong to P_2 but do not belong to P_1. Also, we remove from the process P_1 activities which do not belong to the process P_2. In result, both processes are based on the same set of activities. The preparation is applied not to the processes but to their process matrices. As it was described earlier in this section, such modifications mean adding or removing appropriate rows and columns in the process matrix.

The algorithm is based on A^* search algorithm (e.g., [Pearl 1984]), which finds a permutation while minimizing the distance metric, which is equal to the number of non–equivalent cells. The search starts with an identity permutation $V = 1, 2, \ldots N$. Then, at the depth i, we change the i^{th} element of the current permutation vector and consider all the alternative values for that elements (there are at most (N-1) available alternatives at each moment).

The exhaustive search for a matrix of size $N \times N$ would require $N!$ steps[4], since there are exactly $N!$ distinct permutations of length N. However, the A^* algorithm with a good heuristic can seriously reduce the actual computational time. The heuristic function at depth i is equal to the number of non–equivalent rows or columns in the sub–matrices of size $(N - i) \times (N - i)$ taken from the lower–right corners of both matrices. This heuristic is admissible, since the number of non–equivalent cells in sub–matrices is less then of equal to the number of non–equivalent cells in the whole matrices.

In addition to the pure A^* search, we employ the following improvements to increase the efficiency or our search algorithm:

- Pruning of the intermediate search branches. The principle is the following: at the step i, we consider the number of non–equivalent cells in the upper–left corners of both matrices, and compare this number with the current best

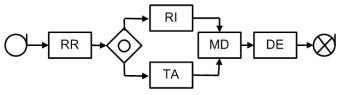
[4]This is the case if we do not allow permutation with repetition. Otherwise, the number of possible options rises up to N^N

result. If the number is greater or equal than the current best result, then it makes no sense to explore the branch anymore.

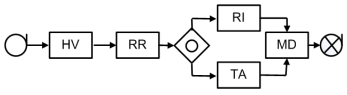
- Iteratively increasing the maximal search depth, and the best result of the previous iteration is used as the initial state for the next iteration.

The cumulative effect of those improvements together with the choice if a heuristic function makes it possible to solve the problem efficiently for the matrices of the size of less than 20, and under reasonable time constraints for the matrices of larger size, as it is shown later in Section 4.2.3.

Example 4.3. *Consider two business processes, shown in Figure 4.7. There the process in Figure 4.7a lacks the activity "HV" at the start of the process, but has and extra activity "DE" at the end, comparing to the process in Figure 4.5b. Process matrices of both processes are also shown in Figure 4.7 below their correspondent processes. Let us find a transformation from the left process to the right one.*



(a)



(b)

	RR	RI	TA	MD	DE
RR	•	⇝	⇝	→	→
RI	•	•	•	→	→
TA	•	•	•	→	→
MD	•	•	•	•	→
DE	•	•	•	•	•

(c)

	HV	RR	RI	TA	MD
HV	•	→	⇝	⇝	→
RR	•	•	⇝	⇝	→
RI	•	•	•	•	→
TA	•	•	•	•	→
MD	•	•	•	•	•

(d)

Figure 4.7: *Example 4.3: Original processes*

First, process matrix for the left process must be normalized. In this case, it means removal of the activity "DE" and adding the activity "HV". The result is shown in Figure 4.8a. The target process matrix is displayed for easy reference in Figure 4.8b. The source matrix can be transformed into the target one by replacing the content of 4 cells in total (the cells in the first row, excluding the upper–left cell).

But it can be also seen that the permutation $(2, 2, 3, 4, 5)$ *transforms the process matrix into the one shown in Figure 4.9a, and this matrix differs from the target one only by one cell.*

	HV	RR	RI	TA	MD
HV	•	•	•	•	•
RR	•	•	⇝	⇝	→
RI	•	•	•	•	→
TA	•	•	•	•	→
MD	•	•	•	•	•

(a)

	HV	RR	RI	TA	MD
HV	•	→	⇝	⇝	→
RR	•	•	⇝	⇝	→
RI	•	•	•	•	→
TA	•	•	•	•	→
MD	•	•	•	•	•

(b)

Figure 4.8: *Example 4.3: Normalized process matrices*

	HV	RR	RI	TA	MD
HV	•	•	⇝	⇝	→
RR	•	•	⇝	⇝	→
RI	•	•	•	•	→
TA	•	•	•	•	→
MD	•	•	•	•	•

(a)

	HV	RR	RI	TA	MD
HV	•	→	⇝	⇝	→
RR	•	•	⇝	⇝	→
RI	•	•	•	•	→
TA	•	•	•	•	→
MD	•	•	•	•	•

(b)

Figure 4.9: *Example 4.3: After transformation*

Once a transformation function is found, it becomes possible to address the task of handling business process evolution, as described in the following section.

4.2.2 Business Process Evolution through Transformation Functions

Consider the situation displayed in Figure 4.1. There is a transformation from the template process to the one of its variants (arrow "1," function f), which can be derived based on those two processes. In the same way, a transformation from the template to the new template can be derived (arrow "2," function g). The main task is to obtain the new variant process in automated way, either following the arrow "3" or "4." Since the result may be different in both cases and there is no way to identify which of them is better, we apply the automated transformation only in the case when the results are the same.

More formally, given two transformation functions f and g and a process A, we call f and g **compatible** w.r.t. a process A if $f \circ g(A) = g \circ f(A)$, (where $f \circ g(A) = f(g(A))$), which means that those transformation can be applied to A in any order.

Such an automated transformation can be beneficial in the case when there are hundreds of variants derived from one template, and one needs to make a quick check if a new template process is compatible with all (or most) of the variants.

The transformation functions f_i which transforms from the original template to the i^{th} variant should be calculated only once and then the result can be stored and reused multiple times whenever a different template process is introduced, thus allowing to make an on-the-fly check of a newly modified template process.

4.2.3 Implementation and Evaluation

In order to evaluate the algorithm, we implemented a proof–of–concept prototype in Java. This prototype is a command line tool which takes as an input two business processes represented in XML Process Definition Language (XPDL) format [van der Aalst 2003], and gives as a result the transformation function between the two input processes. The object of performance test is to evaluate how much time does it take to find a transformation function and does how this time depend on the size of input processes.

In order to get a reasonably large data set, we reused the existing test set "BIT process library, release 2009" [Fahland, Favre, Jobstmann, Koehler, Lohmann, Völzer and Wolf 2009], which contains a set of real business processes. Of them, we extracted a set of pairs of business processes. Two business processes are linked into a pair if they share at least 80% of their activities. In total, we obtained more that 150 pairs, with the size of business processes ranging from 10 to 30 activities.

The tests were conducted on a computer with an Intel® Core™2 Duo processor @3,0GHz, with 3,5GB of RAM, running Java 1.6.0_24. The performance results of experiments are displayed in Figure 4.10. Here X–axis represents the number of activities in the processes, and Y–axis represents the time needed to compute a transformation function in milliseconds. As it can be seen from the chart, for most of the cases the time to compute is bounded by 0.2 second for the processes built of 15–20 activities. The results, however, become worse for the case of 25–30 activities, which is expected behavior for a search problem of exponential complexity.

4.3 Discussion

Formal methods to represent processes provide strong advantages also when managing the evolution of the processes. We have presented a formal language to describe process behaviors and its underlying process models. We have also introduced a way to specify behavioral properties of a business process in matrix form, and how different manipulation with business processes can be represented as manipulations on the corresponding process matrix.

The ways to extend the approach are the subject for future investigation. One line is to study the expressive power and model theory of temporal process logic

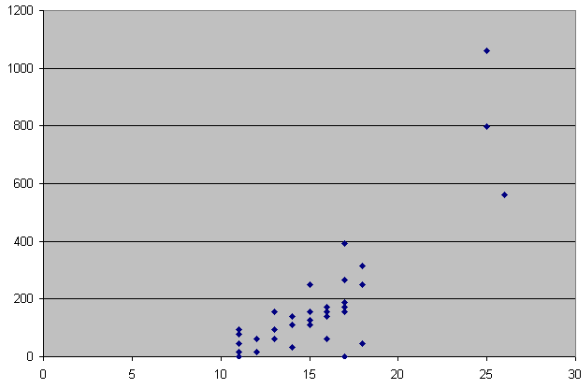


Figure 4.10: *Performance of the Algorithm of Business Process Transformation Extraction*

considering issues such as the completeness of TPL with respect to the Process models and identifying the full axiomatics. The second direction is to increase the expressive power by distinguishing between OR– and XOR–gates. Finally, the algorithm to derive a process transformation can be improved via either introducing better heuristic functions or by analyzing how to apply algebraic matrix transformations for the specific case of process matrices.

Looking into declarative–based solutions described in Chapter 2 [Pesic et al. 2007, van der Aalst and Pesic 2006, Lu et al. 2009, Sadiq et al. 2005] we can draw a conclusion that they are mostly concerned with the task of business process flexibility and the ease of possible change management. The inheritance of business processes which is analyzed in [van der Aalst and Basten 2002, Basten and van der Aalst 2001] solves the problem of connection between a template process and its customized variants. However, it poses additional restrictions on possible customizations, while our approach is applicable to the general case.

The ideas behind annotation–based variability are in line with the task of business process transformation, however, the possible modifications should be anticipated in advance and introduced in the form of annotations or explicit variability points. On the contrary, the ideas of business process transformation presented in this chapter are more general and do not require prior analysis.

The workflow mining techniques [van der Aalst, van Dongen, Herbst, Maruster, Schimm and Weijters 2003] solve the similar problem of discovering a workflow which is represented as a number of entries in the execution log. The specific of our case is that we reuse additional information (i.e. the branching points and their types) which can be extracted from the process definition.

Another way to make an automated process conversion is to merge two busi-

ness process models to obtain a new one, which would inherit the features of both original processes. Although merging of business processes can also be used to deal with business process evolution [Bulanov et al. 2011], the extraction of process transformation function allows to compare the actual change sets and to decide if the modifications are compatible with each other or not.

Temporal planning techniques based on temporal point algebra [Vilain and Kautz 1986, Ghallab et al. 2004] also offer formal temporal–based relationship. While comparing to TPL, all its three base relations ($\rightsquigarrow, \rightarrow, \rightrightarrows$) are represented as only one "less than" relationship in the terms of temporal point algebra.

Finally, formal methods to compare business process models were studied in [Dijkman, Dumas, van Dongen, Käärik and Mendling 2011] with the ability to compare two business process models basing on the similarity metrics. The transformation functions described in this chapter could also be used as a source for a similarity metric, although this was not the main focus of our study.

Published as:
N.R.T.P. van Beest and P. Bulanov and J.C. Wortmann and A. Lazovik – *"Resolving Business Process Interference via Dynamic Reconfigurationv,"* Int. Conference on Service-Oriented Computing (ICSOC–2010), LNCS 6470, pp. 47–60, 2010.
N.R.T.P. van Beest and E. Kaldeli and P. Bulanov and J.C. Wortmann and A. Lazovik – *"Automatic Detection of Business Process Interference,"* 1st International Workshop on Knowledge-intensive Business Processes (KIBP), 2012.
N.R.T.P. van Beest and E. Kaldeli and P. Bulanov and J.C. Wortmann and A. Lazovik – *"Automated Runtime Repair of Business Processes,"* Submitted to Information Systems, 2012.

Chapter 5

The Case of Run Time

Until this point, we discussed the issues of business process variability at design time. In other words, the variations of a business process model were driven by the changes in requirements and therefore had to be reflected as a permanent change in a model. However, the problem of business process variability at run time could be analyzed in a similar manner. The prominent feature of run–time business process variability is that the modifications are usually temporary and address the specifics of one particular case. On the other hand, each of those cases is unique and might never happen again, which makes the number of possible variations to grow, and as a result the major achievements of design time variability become practically useless for the case of run time.

In this chapter, we provide the result of joined work in cooperation with other member of our research group. The idea of using of data–aware dependency scopes to facilitate runtime reconfiguration was initially introduced by Nick van Beest [van Beest, Szirbik and Wortmann 2010], and then further improved in the terms of formal description and implementation in [van Beest, Bulanov, Wortmann and Lazovik 2010]. Later, Eirini Kaldeli brought us the benefits of automated planning being applied to cover the problem of generation of one–off business process variants, and finally we put this all together in a form of a business process orchestration framework which is capable of automated business process reconfiguration at the execution time [van Beest et al. 2012].

The detailed description of the problem of business process interference and data awareness can be found in the work of Nick van Beest [van Beest, Bulanov, Wortmann and Lazovik 2010, van Beest, Szirbik and Wortmann 2010]. The problems of application of automated planning, as well as the detailed description of the Plan-

ner which is mentioned in this chapter are discovered in the work of Eirini Kaldeli [Kaldeli et al. 2011, Kaldeli, Lazovik and Aiello 2009]. In this chapter, the results of our joint work on runtime business process variability are presented, and the contribution of the author of this thesis mostly concerns the Sections 5.2, 5.5, and 5.3 in the parts related to the architecture, implementation, and formal description, respectively.

5.1 Runtime Variability using Dependency Scopes

In the case of design–time variability, the need to create a new variant is induced by the changes in the initial requirements which govern a specific business process. The specific feature of runtime business process variability is that the exact moment to create a new business process variant has to be identified on the fly, while a business process is still running. The reason to do such an early identification is to avoid the execution of the steps which are already known to be unnecessary or even wrong under new circumstances.

In [van Beest, Bulanov, Wortmann and Lazovik 2010], we provided a run-time mechanism which uses dependency scopes and intervention processes to facilitate run–time business process reconfiguration. A *dependency scope (DS)* specifies a part of the BPs whose correct execution relies on the accuracy of a *volatile* process variable, i.e. a process variable that can be changed externally during the execution of the process. If a volatile variable is externally modified while the execution flow resides within the range of the respective DS, an *intervention process (IP)* is triggered as a response, with the purpose to address the modifications in business process structure [van Beest, Bulanov, Wortmann and Lazovik 2010].

By using DSs, testing for potential run–time deviations at each activity can be avoided. As a result, the process designer does not need to foresee all potential process deviations in advance.

However, a significant amount of manual specification of the intervention patterns is still required, since the appropriate IPs may differ considerably depending on the current execution state at which modification of a volatile variable occurred. For complex processes with numerous activities, it is very difficult and time-consuming to define IPs at design–time, as the amount of potential IPs may be particularly high. In addition to that, it cannot be ensured that *all* important variants of the business process are taken into account. Moreover, as the same BP may be deployed and used by more than one organization, different intervention processes have to be specified for each potential BP variant at each organization.

The workload due to extensive manual configuration can be significantly re-

duced by automating the task of IP generation. In the scope of this chapter, domain-independent AI planning is employed to automatically generate IPs, which restore the consistency of a BP. In that way, the manual work required by the domain designer is reduced to the specification of a high-level goal, which describes in a declarative way the desired consistent state that has to be reached in case of interference. To realize such a level of automation, additional semantic annotations are required, which capture the functional aspects of the activities participating in the business domain in terms of preconditions and effects, in spirit with existing process ontologies such as OWL-S [World Wide Web Consorcium (W3C) 2004].

5.1.1 Dependency Scopes within WMO Process Example

In order to guard for changes of the volatile process variables, Dependency Scopes (DSs) can be defined covering a section of the process for which such a change poses a potential risk of interference. We consider the case study which is described in Section 1.1.1. In Figure 5.1, a part of the process from Figure 1.4 is annotated with the appropriate DSs. The section covered by DS1 relies on the accuracy of the address as well as the medical condition of the citizen, while the section covered by DS2 relies on the accuracy of the WMO eligibility criteria. That is, if the legal criteria that are relevant for the used contract have changed, this might affect the order itself, or the potential suppliers that are participating in the tender procedure. Finally, the section within DS3 depends on the address and the medical condition of the citizen as well. However, it is separate from DS1 because of the syntax of the BP.

If a DS is triggered by an external change on its process variable, potentially some recovery activities need to be executed to restore consistency. This leads to intervention processes, to be discussed in the next subsection.

5.1.2 Required Intervention Processes

The required IPs may differ for each situation. Let us consider for example the DS1 of Figure 5.1. If the provision concerns the delivery of a wheelchair, and the address change is detected before the order for the wheelchair is sent to the supplier, the following actions have to be performed. First, a new home visit to the new address has to take place in order to check the new residence and living conditions, which are important for the advice provided by the external consultant. Then, the medical expert has to provide an updated advice, taking into account the characteristics of the new residence, and then a new decision has to be made by the municipality considering the newly acquired information (if for example the user has moved to a nursery home, then the citizen may no longer be eligible for a wheelchair). If

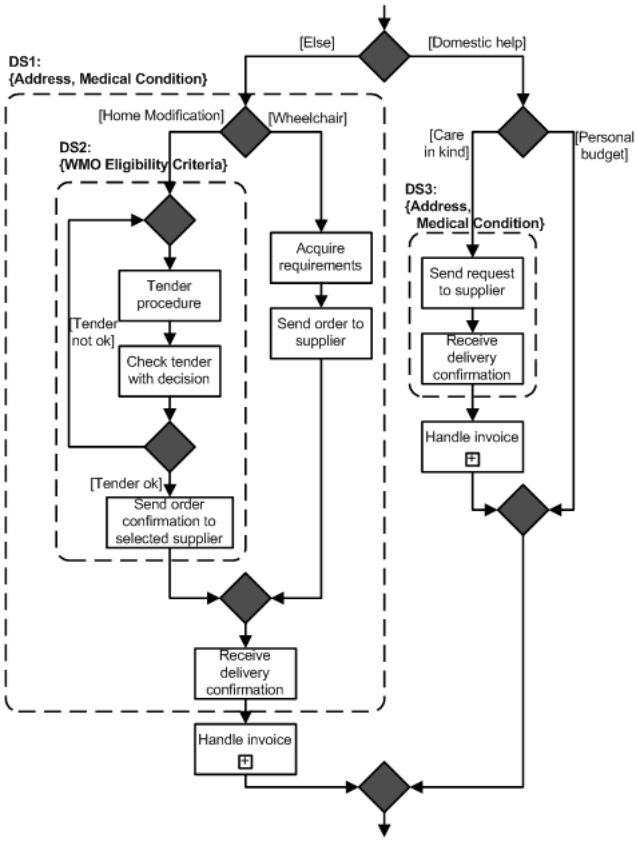


Figure 5.1: *Dependency scopes in the WMO process.*

the municipality still approves the citizen's request, the requirements concerning the wheelchair have to be updated, and the respective order has to be sent to the supplier, as shown in Figure 5.2a. However, if the order was already sent to the supplier before the new information became available, this order has to be canceled prior to proceeding with the new one (Figure 5.2b). It should be noted, that if there is an operation that offers the possibility to update the contents of an order that has already been issued (given that it has not already been delivered), the IP would include this operation rather than cancelling the existing order and re-issuing a new one. After the execution of the appropriate IP, the process proceeds from the state just after the DS.

Similarly, in case of a home modification, the form of the appropriate IP depends on the state at which the address change has occured as well. If the address changes before the order is sent, it is sufficient to execute the IP as represented in Figure 5.2c.

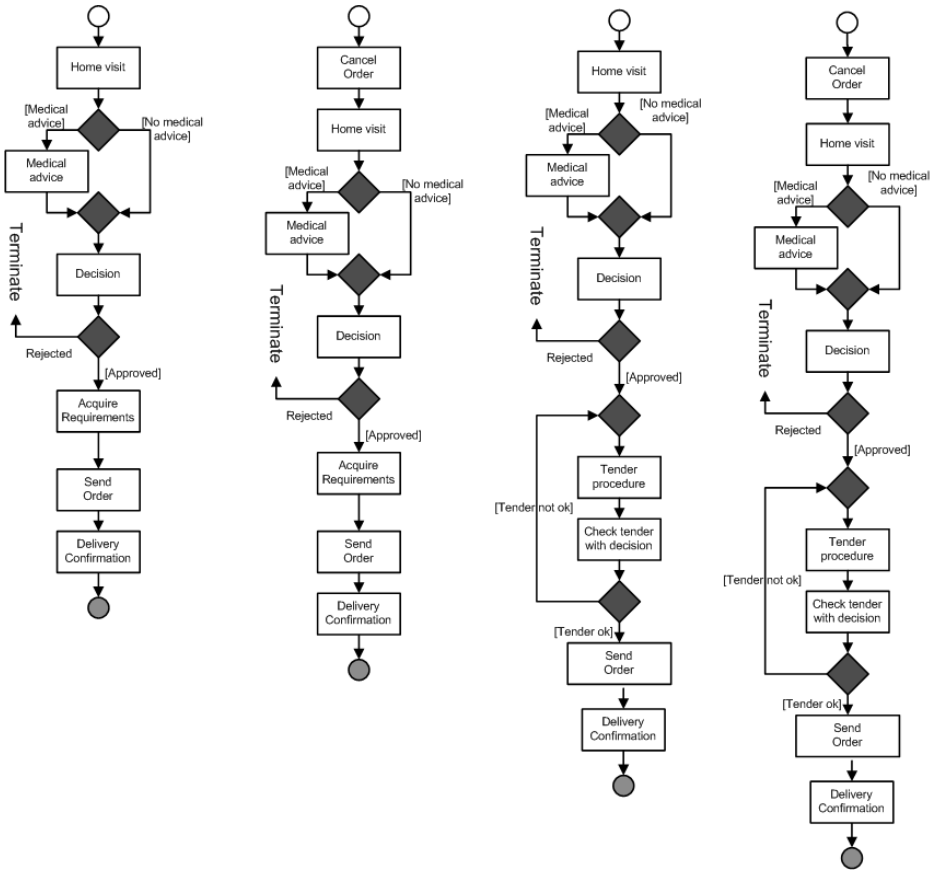


Figure 5.2: *Required intervention processes corresponding to DS1, in case of an address change*

Since the specifications on the order for a home modification directly rely on the physical properties of the house, a change of address implies a cancellation of the order if an order has already been sent, as shown in Figure 5.2d. However, these examples assume that the citizen moves *within* the municipality (in our example this is 'Groningen'). If the citizen has moved to another municipality, the order should be cancelled and a notification sent to the city hall. Then, the entire process should be aborted, regardless of the requested provision, as each municipality has its own policies and procedures (Figure 5.2e).

It becomes evident from the example that even for a *small DS*, *the complexity and workload required for specifying the IPs is high*. Addressing the consequences of an address change *on a small part of the process* requires 5 distinct IPs. Anticipating and

manually specifying the appropriate IP is difficult, time-consuming, and prone to oversights of possible situations that may arise: different IPs are required not only depending on the current state, but also on the actual value of the modified variable. As a result, for each possible state in a DS and type of change to the modified variable, a different IP may be required. Moreover, since the same BP may be used by more than one municipality, different IPs have to be specified for each of the different cases, as they may have access to different compensation services or comply with different rules.

5.1.3 Automatic Intervention Process Generation

Instead of relying on a procedural specification of IPs (or equivalently a composite IP with a huge number of conditional branches to take into account all possible combination of situations), we propose to assign the task of computing the appropriate IP to an AI domain-independent planner. The task of the BP designer in this case is reduced to a declarative specification of the properties that have to be fulfilled at a higher level of abstraction, considering some general, more intuitive cases. This way, it is not necessary to explicitly specify how these properties can be achieved under all possible combinations of environmental conditions and execution states of the BP. In this case the desired properties are captured by a *goal*. The goal represents the desired state after the end of the DS, along with some (optional) features to be achieved. For example, considering the case of DS1 where the address change indicates that the user has moved to a new home within the range of the municipality, all that has to be captured by the goal is that at the end "the order of the citizen has to be delivered." The goals accompanying DS1 are presented in Section 5.3.2. In this respect, the BP designer does not have to be concerned with *which* service operations are available, whether the order concerns a wheelchair or home modification, whether the order has already been sent out or not, which actions have to be performed again and which not, as well as the order of these actions.

The approach adopted herein, leaves it to an AI planner to automatically generate the IP whenever possible, based on a semantically enriched services and BP specification, the current state and the value of the volatile process variable. The assumption is that appropriate semantic annotations are available: the semantics accompanying the pool of services used by the BP have to be specified once and are reused by different BPs, while the BP-specific semantics represent the BP structure in a BPEL-like way, along with the direct dependency of some variables on the validity of some other variables (see Section 5.3). In the next section, the architecture of the framework supporting the automatic generation of IPs is presented, and the interplay between the different constituent modules is explained.

5.2 Architectural Overview

Figure 5.3 provides an overview of the main components of our framework, along with their basic interactions. A *Process Modeller* (PM) is used to assist with the task of the graphical modeling of the BP, providing a selection of standard control blocks like sequence, flow, XOR etc., and design tools for modeling DSs, in accordance with their definition provided in Section 5.3. DSs include the specification of some high-level *goals* of declarative nature, which have to be fulfilled by the respective intervention process.

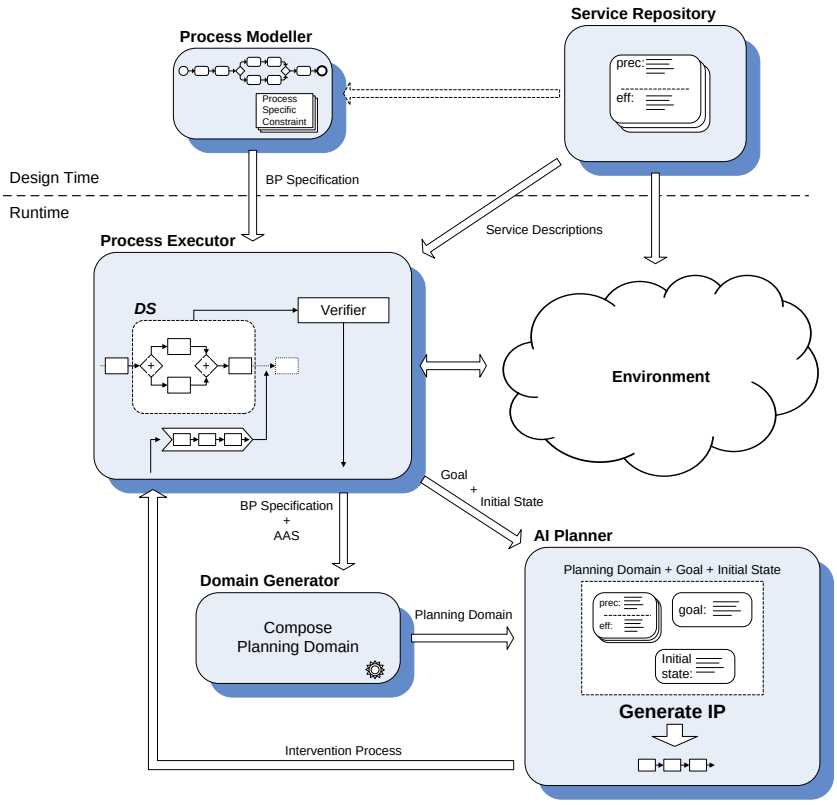


Figure 5.3: *Main components of the framework and their basic interactions*

The BP modelled by the PM uses activities that are available in the *Service Repository* (SR) by means of service operations. The SR keeps a list of service instances (providers) that offer a set of service operations. Each service instance implements a service description, which specifies the interface of the service annotated by some extra semantics. These semantics allow each service operation to be represented as

a planning *action*, reflecting its functional behaviour in terms of preconditions and effects, which are necessary for enabling the automatic generation of intervention processes. A subset of the service operations are referenced by the BP specification, whereas operations offered by other service instances can be marked as pertinent compensation actions, and can become part of an IP if necessary.

The *Process Executor* (PE) is responsible for executing the BP step by step (i.e. the normal course of events as specified during design-time), and takes care of discovering, binding and invoking the respective service operations residing in the *Environment*, according to their specification as included in the SR. Some of the variables describing the state of the environment can be directly changed by the process being executed by the PE, through the invocation of services it has access to, or can be modified by some external process. In the latter case, the PE receives a modification event, and updates its current internal state accordingly. In addition to process execution, the PE supports the use of DSs. Before execution of each activity, the PE checks whether the current state indicates a modification of the volatile variables that are guarded by a DS that covers this activity. If so, it verifies whether any of the conditions specified in the DS hold. If a condition holds (e.g. the new address is outside the current municipality), then the PE interrupts the execution and invokes the *AI Planner*. The AI Planner requires as input (i) the Planning Domain (ii) the initial planning state (i.e. the values of all process variables at the current execution step and a set of variable interdependencies), and (iii) the goal describing the desired properties to be achieved (e.g. a notification should be sent to the city hall). Before explaining the AI planner in more detail, we will discuss the notion of a Planning Domain.

The Planning Domain is computed by the *Domain Generator* (DG) only once per BP, the first time that the PE identifies the need for automatic IP generation. In order to form the Planning Domain, the PE passes the *Atomic Actions (AA)* and the BP specification (provided as output by the PM) to the DG. The AA represent the BP-pertinent action descriptions as kept in the SR (i.e. the ones referenced by the BP along with the compensation operations).

Given the Planning Domain, the initial state and the goal, the AI planner generates the appropriate IP that achieves the associated goal. The generated IP is then returned to the PE. After the execution of the IP, the PE either proceeds with the execution of the original BP, starting from the state right after the triggered DS (as in Figures 5.2a-d, where the original BP execution resumes after ”Delivery”), or aborts if the IP leads to a state that indicates the termination of the BP (as in Figure 5.2e). If the former is the case, potential branches that were running in parallel are also resumed from the point they were interrupted, otherwise the entire process is interrupted. In the case of nested DSs, as for example DS1 and DS2 of Figure 5.1, the

PE checks first whether the conditions specified by the outermost DS are true, and if not, it proceeds by checking the inner DS. The generated IP is executed within the scope of the DS it was triggered from and the parent DSs, i.e. variable modifications that are received during the execution of an IP are covered by the same set of DSs that covered the action before which the planner was fired. If no plan can be found, i.e. there is no way to overcome the inconsistencies caused by the volatile variable modification using the activities it has access to, then the BP is canceled, and a request for manual inspection is issued.

The implementation of the PE and the PM is presented in Section 5.5, while more details about the AI planner are provided in Section 5.3.3 and Section 5.4.

5.3 Basic Concepts

In this section, the definition of the basic concepts is provided, where the approach for BP repair is built upon.

5.3.1 Business Process

First, we define the SR, which comprises a set of service descriptions and a set of service instances, which "implement" some service description. The service descriptions comprise semantics, which specify the functionality provided by a service type. The service instances specify the way to invoke a certain service conforming to a service description. The semantic markups defined in the service description are necessary in order to automate the task of IP generation. They are expressed in terms of *preconditions*, which model the propositions that have to hold in the current state for an activity to be executed, and *effects*, which formulate how variables are changed by the activity's execution. The service descriptions are based on an IOPE (Input Output Preconditions Effects) model, which is followed by established Web Service semantic languages like WSDL-S [World Wide Web Consorcium (W3C) 2005] and OWL-S [World Wide Web Consorcium (W3C) 2004].

Definition 5.1 (Service Repository (SR))**.** *A* **Service Repository** $SR = (SD, SI)$ *is a storage, which keeps a set of* Service Descriptions SD*, and a set of* Service Instances SI*. A Service Description* $sd \in SD$ *is a tuple* $sd = (sdid, O, SV)$*, where* $sdid$ *is a unique identifier,* O *is a set of service operations, and* SV *is a list of variables , each ranging over a finite domain. These variables correspond to state variables internal to the service, whose value can be changed by the operations of the service. Each service operation* $o \in O$ *is a tuple* $o = (id(o), in(o), out(o), prec(o), \mathit{eff}(o))$ *where*

- $id(o)$ *is the identifier of the operation*

- $in(o)$ *is a list of variables that play the role of input parameters to* o*, ranging over finite domains*
- $out(o)$ *is a list of variables that play the role of output parameters to* o*, ranging over finite domains*
- $prec(o)$ *is a set of preconditions and* $eff(o)$ *a set of effects, as defined in Definition 5.4 with* $Var = in(o) \cup out(o) \cup SV$

A Service Instance $si \in SI$ *is a tuple* $si = (iid(si), st(si))$*, where:*

- $st(si)$ *is the unique identifier of the service description* $sd \in SD$ *this instance complies with*
- $iid(si)$ *is an instance identifier. For each pair of service instances* $si_1, si_2 \in SI$ *that have the same service description identifier* $st(si_1) = st(si_2)$*,* $iid(si_1) \neq iid(si_2)$*.*

The SR plays the role of a pool service descriptions and instances, which are used as the building elements of different process specifications. In the following, the definition of a Business Process (BP) is provided, which includes the basic activities and control structures such as sequence, flow and XOR. The BP is enriched with DSs, which also constitute parts of the process. The syntax of the BP is well-defined and unambiguous, so that they can be directly executed by the Process Executor (see Section 5.5.2) and automatically transformed to a representation usable by the planner. The BP definition used in this chapter is block-structured using the standard XOR, flow and sequence constructs, in spirit with BPEL's notation. This definition is not in line with a BPMN–based definition employed in Chapters 3–4, because the block–structured one is more suitable for the management of running business process instances. As shown in [Ouvans, Dumas, ter Hofstede and van der Aalst 2006, Kopp, Martin, Wutke and Leymann 2008], a BP representation following a graph-based model, such as the BPMN-like notation used in Figure 1.4, can be easily mapped to BPEL-like block structures, similar to the ones used in our definition. The representation is ultimately a tree structure where a block can have other blocks as children, and for each block its parent can be obtained. The definition is recursive, so that control structures and DSs can be nested within each other.

Definition 5.2 (Business Process (BP))**.** *Given a Service Repository* SR=(SD, SI)*, a* **Business process** *is a tuple* $BP = (PV, E)$*, with* E *being a process element* $E = (act \mid seq \mid flow \mid XOR \mid repeat \mid while \mid DS)$*, where:*

- $PV = PV_i \cup PV_e$ *is a set of variables ranging over finite domains.*

  - PV_i *is a set of internal variables, which are declared at the BP level (BP-specific). A subset of these variables are passed as input parameters to the entire BP, in which case we write* $BP(pv_1, \ldots, pv_n)$*, where* $pv_i \in PV_i$ *and* pv_i *can be initialized with specific values at execution time.*
  - PV_e *is a set of external variables, which refer to variables declared in the SR. An external variable* $v \in PV_e$ *is a reference* $sdid.iid.fid$*, with* $sdid$ *being the identifier of a service description* $sd = (sdid, O, SV) \in SD$*,* iid *the identifier of a service instance*
    $si = (iid, sdid) \in SI$*, and* fid *the identifier of some state variable (field)* $f \in SV$*.*

- act *is a process activity, which represents the invocation of an operation that exists in* SI*. It is a tuple* $act = (id(act),\ in(act), out(act))$*, where* $id(act)$ *is a reference* $sdid.iid.oid$*, where* $sdid$ *is the identifier of a service description* $sd = (sdid, O, SV) \in SD$*,* iid *is the identifier of a service instance*
  $si = (iid, sdid) \in SI$*, and* oid *is the identifier of some operation* $o \in O$*. The input and output parameters of* act *refer to the input and output parameters of the respective* oid*, i.e.* $in(act) = in(oid)$ *and* $out(act) = out(oid)$*. The input (output) parameters of all activities in the BP form the sets* $IP\ (OP)$*. Input variables can be assigned with constant values or other process variables. We thus write an action invocation as* id(act) (ip_1:=v_1, …, ip_n:=v_n)*, where* $ip_i \in in(act)$*,* $v_i \in (PV \cup OP)$*, or* v_i *is a value compliant with* ip_i*'s domain. There are also two extra special types of activities:* $no\text{-}op$*, which represents an idle activity with* $true$ *preconditions and no effects, and* $terminate$*, whose execution causes the whole BP to halt. Directly after an action invocation, an action's output can be stored in some process variable, in which case we write* $(pv_1 := op_1, \ldots, pv_n := op_n)$*, where* $op_i \in out(a)$ *and* $pv \in PV_i$*.*

- seq *refers to a totally ordered set of process elements, which are executed in sequence. The following notation is used:* $seq\{e_1 \ldots e_n\}$*, where* e_i *is a process element.*

- $flow$ *represents a set of process elements, which are executed in parallel. We write* $flow\{e_1 \ldots e_n\}$*, where* e_i *is a process element.*

- XOR *is a set of tuples* $\{(c_1, e_1), \ldots, (c_n, e_n)\}$*, where* e_i *is a process element and* c_i *is a logical condition* C*, which conforms to the following syntax:*
  $C \quad ::= prop | \wedge_j\ C_j | \vee_j\ C_j) | \neg C_j$
  $prop ::= var \circ value \ \mid\ var1 \circ var2\ \ |(var1 \diamond var2) \circ value,$
  where $var, var1, var2 \in (PV \cup OP)$*,* $value$ *is some constant belonging to* var*'s domain,* $\circ$ *is a relational operator (*$\circ \in \{=, <, >, \neq, \leq, \geq\}$*) and* $\diamond$ *a binary operator (*$\diamond \in \{+, -\}$*). All* c_i *participating in an* XOR *are mutually exclusive, i.e. for any given assignment to* $PV \cup OP$*, only a single* c_i *evaluates to true, and* e_i *will be executed if* c_i *evaluates to true. We write* $XOR\{c_1 \Rightarrow e_1, \ldots, c_n \Rightarrow e_n\}$*.*

- *repeat represents a loop structure, and is defined as a tuple* $(pe, c\{pe_i\})$*, where* c *is a logical condition as already defined, and* pe, pe_i *are process elements.* c *is evaluated just after the end of* pe*, and if it holds then* pe *is repeated, after the execution of the optional* pe_i*. We write* $repeat\{pe\}\ while(c\{pe_i\})$*, with* $\{pe_i\}$ *being optional.*

- $while$ *is similar to* $repeat$*, with* c *being evaluated before* pe *starts.*

- *S is a dependency scope as defined in Definition 5.3.*

5.3.2 Dependency scope

The DS is a **guard-verify** structure, where the critical part of the BP is included in the *guard* block, while the *verify* block specifies the types of events that require intervention. Whenever such an event occurs, the control flow is transferred to the *verify* block, and the respective goal is activated. Once the resulting IP finishes execution in the updated environment, the control flow of the BP continues from the point following the *guard-verify* structure, unless it is explicitly forced to terminate.

Definition 5.3 (Dependency Scope (DS))**.** *Given a* $SR = (SD, SI)$ *and a* $BP = (PV_i \cup PV_e, E)$*, a* **dependency scope** *is a tuple* $DS = \langle guard(VV)\{E_g\},\ verify(\{(case(C_i) : G_i \mid E_{ip} \mid terminate(G_i) \mid terminate(E_{ip}))\})\rangle$*, where:*

- $guard(VV)$ *indicates the set of volatile variables* $VV \subset PV_e$ *whose modification triggers the verification of the DS, and* E_g *a process element in the* BP*. Whenever during the execution of* E_g *an event indicating a change in the value of a volatile variable* $vv \in VV$ *is received, the* $verify$ *part of the DS is triggered, and* BP*'s execution is interrupted.*

- $verify(\{(case(C_i) : G_i \mid E_{ip})\})$ *comprises a set of tuples consisting of a case-condition* C_i *and a goal* G_i *or a process element* E_{ip} *to be pursued if* C_i *holds.*

  - C_i *is a logical condition, as defined in Definition 5.2. Providing a case condition is optional, with the default interpretation being* $C_i = TRUE$*.*

  - G_i *specifies a goal, which ensures the satisfaction of the properties that reflect the state right after the final activity of* E_g*.* G_i *is specified in the goal language supported by the planner as presented in [Kaldeli et al. 2009]. After interrupting the BP execution, the plan that satisfies the respective* G_i *(if it can be found) is executed. When the plan's execution is completed, the* BP *is resumed at the state after* E_g *and from any other parallel branches of the* BP *that were interrupted.*

– *If an E_{ip} is pre-specified to be executed in case C_i holds, then BP's execution is interrupted, E_{ip} is executed, and after its completion BP resumes from the end of E_g.*

- *$terminate(G_i)$ ($terminate(E_{ip})$) forces the process to terminate, i.e. abort the rest of BP's execution, after fulfilling G_i (completing E_{ip}'s execution).*

Following Definition 5.3, the DS specification representing DS_1 of Figure 5.1 is the following:

```
guard(address, medCond){
 seq{
  XOR{
   ... \* subprocess *\
  }
  receiveDeliveryConf(dlIn_orderId=orderId, dlIn_cid=bpCid,
   dlIn_address=orderAddress,dlIn_delContents=orderContents)
 }
} verify{
 address.county ≠ 'Groningen':  terminate(achieve-maint
   (notifiedCityHall('countyChange')=TRUE ∧ invalid(orderId)))
 address.county = 'Groningen' AND medCond ≠ deceased:
  achieve-maint(known(dlOut_conf))
  medCond='deceased' :  terminate(achieve-maint(invalid(orderId)))
}
```

According to DS_1, if a modification in the address or the medical condition occurs within the scope of the guarded subprocess, the following goals are pursued:

- If the address change indicates that the citizen has moved to another municipality, the goal ensures that the intervention plan leads to a state, where the order for a wheelchair or home modification (depending on the value of the "provision" variable, which is determined by the activity "Intake and Application") has been cancelled, and a respective notification is sent to the city hall. The plan will be equivalent to IP (e) of Figure 5.2.

- If the medical condition has changed to some new value that does not indicate "deceased" and the customer is still within the range of the municipality, the final desired state is that the delivery of wheelchair or home modification is performed by taking into account the new situation (the new medical condition and/or address). Depending on the state at which the modification occurs

and the kind of the modification, the generated plan is one of the IPs (a) to (d) of Figure 5.2. After the plan's execution the BP execution resumes to handle the invoice.

- If the new value of medical condition indicates "deceased," then the goal specifies that the order should be invalidated.

Depending on the state of the DS in the original BP, at which the relevant volatile variable modification was identified, the generated plan may vary considerably for the same goal. This way, one DS definition covers all forms of IPs specified in Figure 5.2, which are generated automatically by the planner. The domain designer just prescribes in the goal what properties have to be satisfied during recovery, but is not required to know the combinations of actions that can achieve the goal. The planner uses a heuristic that promotes optimal plans. As a result, the planner may come up with different plans that fulfill the goal, depending on the available services. Considering, for example, an address change after an order has been sent in DS1 in Figure 5.1. If the supplier service offers an $updateOrder$ operation, the planner will advocate an update in the order address information, instead of cancelling the existing order and sending a new one.

Interdependencies between variables are also defined on top of the BP specification, prescribing the direct dependency of some variables on the validity of some other variable. The $dependsOn$ relation is used for this purpose:
$dependsOn(v) = \{v_1, \ldots, v_n\}$. Whenever a change in variable v is discovered or whenever v is invalidated (by transitivity, as an effect of some other variable interdependency) by the PE, the direct invalidation of the current values of $v_1, \ldots, v_n$ is automatically implied, without the need of some special-purpose process to take care of that. For example, *dependsOn(bpAddress_address)* = {*hvOut_homeInfo*}, since $hvOut\_homeInfo$ refers to the information retrieved for the specific $hvIn\_address$. Thus, if the person moves to some other address, the collected information is not valid anymore. In turn, a set of variables, like $arOut\_requirements$ reflecting the acquired requirements concerning the wheelchair, are directly dependent on $hvOut\_homeInfo$. On the other hand, an $orderId$ is not directly dependent on the address, since it remains valid after these variables change, unless some other course of interaction actively cancels it. These additional statements are of particular relevance when the change of a volatile variable is discovered, so that all information directly dependent on the consistency of the volatile variable also becomes obsolete, as shown in Section 5.4.2.

5.3.3 The Planning Domain

The PE constructs a planning domain given a BP specification and a SR, which is used by the planner for generating the IPs upon recovery requests. In this subsection, a simplified definition of a *Planning Domain (PD)* is provided, in line with [Kaldeli et al. 2011]. The planning domain has some special characteristics that distinguish it from classical planning representations. The domain accommodates for numeric fluents, which can range over finite domains, including the input arguments of actions. The planning domain is enriched with a knowledge-level representation to model observational actions (sensing effects), which is useful for dealing with XORs with conditions on the outcome of such actions. The automatic composition of a planning domain is further described in Section 5.4.

Definition 5.4 (Planning Domain (PD))**.**
A **Planning Domain** *is a tuple* $\mathcal{PD} = \langle Var, Par, A \rangle$*, where:*

- Var *is a set of variables. Each variable* $v \in Var$ *ranges over a finite domain* D^v*.*
- Par *is a set of variables that play the role of input parameters to members of* A*. Each variable* $p \in Par$ *ranges over a finite domain* D^p*.*
- A *is the set of actions. An action* $a \in A$ *is a triple* $a = (id(a),\ in(a), precond(a), effects(a))$*, where:*
  - $id(a)$ *is a unique identifier*
  - $in(a) \subset Par$ *are the input parameters of* a
  - $precond(a)$ *is a propositional formula over* $Var \cup Par$*;*
  - $effect(a)$ *is a formal description of the outcome of the action* a*, including the description of the output parameters.*

The effect of a given action may be either deterministic or non–deterministic. The former case means that the outcome is known in advance provided the values of all of the input parameters. The latter case, on the contrary, means that the outcome cannot be anticipated and the only way to know the result is to actually invoke the corresponding action. Non–deterministic effects are useful in deferred choices, i.e. XOR–splits here the condition depends on some interaction with the actual execution environment. Its verification is thus deferred until runtime, after some variable is determined during the execution of a knowledge–providing action.

The domain is extended with additional variables to model the knowledge–level representation, and to distinguish between sensing and world–altering actions. These variables are generated automatically given a planning domain $\mathcal{PD}$. First, for

each $var \in Var$, a new boolean variable var_known is introduced, which indicates whether var is known at state s ($var\_known(s) = true$) or not ($var\_known(s) = false$).

Following a common practice in many planning approaches, we consider a *bounded* planning problem, i.e. we restrict our target to finding a plan of length at most k, for increasing values of k. Considering a planning domain extended with the knowledge–level representation $PD = \langle V, A \rangle$, the target is to encode PD into Constraint Satisfaction Problem (CSP). The reason behind that transformation is to represent our problem in the form which is acceptable by one of the standard constraint solvers.

First, for each variable $x \in V$ ranging over D^x, and for each $0 \leq i \leq k$, we define a CSP variable $x[i]$ in CSP with domain D^x. Actions are also represented as variables: for each action $a \in A$ and for each $0 \leq i \leq k-1$ a boolean variable $a[i]$ is defined. This way, the computed plan can include parallel actions, which may save time during the execution. Action preconditions and effects, are automatically encoded as constraints on the CSP state variables, based on the formulation described in [Ghallab et al. 2004].

5.4 Automatic Intervention Process Generation

In this section, the preliminary steps required for IP generation are explained. These steps comprise the generation of a planning domain by the DG and composition of the initial planning state by the PE.

5.4.1 Generation of the Planning Domain

The semantic specifications stored in the Service Repository are process-independent, and capture the generic functionality of the respective service operations in terms of preconditions and effects, so that they can be used in the context of various BPs. Usually these preconditions and effects concern the set of inputs and outputs of the respective operations and some additional aspects that are internal to the particular service.

For each BP, the operations of a subset of service instances in the Service Repository are marked as pertinent compensation methods. These methods can be part of the intervention processes for repairing the BP, and are annotated by the domain designer. If a permissive approach is adopted, the entire set of service instances in the SI part of the SR is allowed to be used by the IP. These compensation methods, along with the invocation methods referenced by the activities in the BP, form the

BP-pertinent Methods (BPPM) set. For each method $sdid.iid.oid \in BPPM$ of a service instance $si = (iid, sdid) \in SI$, whose service description includes an operation o with $id(o) = oid$, the PE generates some instance-level variables, preconditions, and effects, based on its iid and the operation description o this method realizes. The resulting set of instance-level method descriptions forms the *Planning Domain*, as described below.

Given a Service Repository $SR = (SD, SI)$, a BP, and a set of BP-pertinent Methods *BPPM*, the Planning Domain is built in the following way:

- When the PE receives a request to execute the BP, a unique instance reference $bp\text{-}iid$ is assigned.
- For each method $bpo = sdid.iid.oid \in BPPM$, the service description $sd = (sdid, O, SV) \in SD$ is found, and the operation $o = (id(o), in(o), out(o), prec(o), \mathit{eff}(o)) \in O$ with $id(o) = oid$ is retrieved.
- For each input parameter $ip_i \in in(o)$, a new input variable is created for $sdid.iid.oid$, with name $bp\text{-}iid.sd.iid.oid.ip_i$ and a domain identical to ip_i. Similarly, for each output parameter $op_i \in out(o)$, a new output variable is created, with name $bp\text{-}iid.sd.iid.oid.op_i$ and a domain identical to op_i. The resulting instance-level input and output parameters form the sets $in(bpo)$ and $out(bpo)$ respectively.
- Based on the preconditions and effects of o, the sets $prec(bpo)$ and $\mathit{eff}(bpo)$ are generated, by substituting each input and output parameter with name v appearing in $prec(o)$ and $\mathit{eff}(o)$ by the reference $bp\text{-}iid.sdid.iid.oid.v$. In case of a service state variable $var \in SV$ with local name v, the reference is substituted with the universal name $sdid.iid.v$, which is BP independent. If $sdid.iid.v$ has not been met before, the respective variable with name $sdid.iid.v$ and with domain identical to var is created.

This way, for each $act = sdid.iid.oid \in BPPM$ the invocation method description tuple $imd = (bp\text{-}iid.sdid.iid.oid, in(act), out(act), prec(act), \mathit{eff}(act))$ is created by the PE. Each imd is converted to a planning action (as defined in Definition 5.4) $a = (id(a) = (bp\text{-}iid.sdid.iid.oid,\ in(a_i) = in(act)),\ prec(a) = prec(act),\ \mathit{eff}(a) = \mathit{eff}(act))$.

The planning domain which is formed as described above reflects only the atomic-level semantics of the actions. In the context of a certain BP, the universal action descriptions have to be enriched with extra preconditions and/or effects, which reflect the process-specific interdependencies, and which can be automatically inferred from the structure of the BP. The details of the capturing of such process–level details can be found in [van Beest et al. 2012].

5.4.2 Composition of the initial planning state

The initial planning state comprises the values of all variables at the current state of execution and the knowledge level with respect to the variables interdependency rules. Given the manually specified variable interdependencies in terms of the $dependsOn$ sets, these are enriched during execution of the BP by the PE: if an action comprising an assignment effect $assign(v', v)$ or an increase(decrease) effect $increase(v', v)$ ($decrease(v', v)$), has been executed, variable v' is added automatically to the $dependsOn(v)$ set (if the set does not already exist, it is created). Each time the AI planner is called by the PE, the initial planning state is formulated as follows.

- Each variable $var \in PV$ is equal to a value corresponding to the state of execution, i.e. considering the assignments to the BP input parameters, the outputs of the service invocations, the assignments to variables, and the received external events (for more details see Section 5.5).
- For each variable var for which no specific value has been acquired yet, the respective knowledge variable $known\_var$ is set to false at the initial state ($known\_var(0) = false$).
- Given a change event on a volatile variable vv, the interdependency rules are parsed. For each $var \in dependsOn(vv)$, $known\_var(0) = false$, indicating that the value of var as reflected by the current state of execution is not valid. The same is done recursively $\forall\ var' \in dependsOn(var)$, $\forall\ var \in dependsOn(vv)$.

5.4.3 Generating the IP

By starting from the initial state as delivered by the PE, and depending on the goal, the IP can be computed by the AI planner using the planning domain. This IP may include the re-invocation of activities with the up-to-date input parameters, if this is required to achieve the goal (e.g. pay a visit to the new address to acquire the informed requirements), or try to find a sequence of "undo" actions that actively lead to the invalidation of some variables (e.g. try to cancel an order that has been sent if possible). In case of deferred choices (i.e. XOR-constructs where the value of a variable participating in the respective condition is unknown off-line) it has to be ensured that the right branch is followed at runtime. One way to address this issue is to rely on conditional plans, as e.g. presented in [Pistore, Marconi, Bertoli and Traverso 2005, Hoffmann, Weber and Kraft 2010]. However, for these approaches it is difficult to deal with sensing outcomes that range over numeric-valued domains. Herein, we resort to a re-planning mechanism to model deferred choices,

where the value of the condition is acquired during runtime. The plan originally returned by the planner is optimistic, i.e. the variables that are unknown off-line are assumed to have values that lead to the shortest plan that fulfills the goal. Thus, in the case of the IP Figure 5.2c, it generates the plan that corresponds to the assumption that the output of "HomeVisit" $hvOut\_maRequired = false$, which indicates that the home inspection does not entail the need for a medical advice, that the decision is positive, and that the supplier selected by the customer is approved. Whenever a knowledge-providing activity is executed by the PE, and the initially unknown variable is instantiated, the outcome is compared with the value assumed by the plan. That is, it is checked whether the new knowledge incorporated in the CSP violates any constraint. If no violation is detected, then the execution of the IP may proceed according to the initial plan. Otherwise the planner is invoked again with the same goal and a new initial state, including the value of the sensed variable. As a result, a request for a Home Modification may require the following series of interactions when planning for Goal *achieve-maint(known(delOut_delId))* (see Section 5.3.2), in order to obtain the IP shown in Figure 5.2c (the input parameters are omitted for brevity):

Initial plan: $\{HomeVisit, Decision, TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute $HomeVisit$ Output: $hvOut\_maRequired = true$, **constraint violation, re-plan**

New plan: $\{MedicalAdvice, Decision, TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute: $MedicalAdvice$	$maOut\_medInfo = `Document12A'$
Execute $Decision$	Output: $dcOut\_approvalCheck = true$
Execute $TenderProcedure$	Output: $tpOut\_tenderSelection = `ACMFrizianConstructions'$
Execute $CheckTender$	Output: $ctOut\_tenderOK = false$, **constraint violation, re-plan**

New plan: $\{TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute $TenderProcedure$	Output: $tpOut\_tenderSelection =$'van der Meer Elevators'
Execute $CheckTender$	Output $ctOut\_tenderOK = false$
Execute $SendOrderToSelSupplier$	Output: $soOut\_orderId = `14578AS'$
Execute $Delivery$	Output: $dlOut\_conf = `Delivered'$

If the output of "Decision" is negative, then no plan exists that satisfies the goal. In that case, the planner returns a message indicating that the goal is not satisfiable, causing the BP execution to be aborted. In total 9 service operations are invoked as part of the IP.

5.5 The Prototype

The proposed approach for automatic process recovery upon data changes has been implemented in a prototype, comprising the components of the architecture outlined in Figure 5.3.

5.5.1 Process Modeller

The Process Modeller (PM) is implemented in Java, by the use of standard Java 2D graphical libraries. It supports all basic BP modelling constructs, including flows, XOR splits etc., with an added support for DS modelling. Furthermore, the PM provides for the declaration of the process variables, i.e. the definition of their name and type. However, the actual object creation is handled by the PE, which keeps and manages a local database as described in Section 5.5.2. The PM is connected to the Service Repository, so that the BP designer can use service operations that exist in the SR as activities in the BP being modelled.

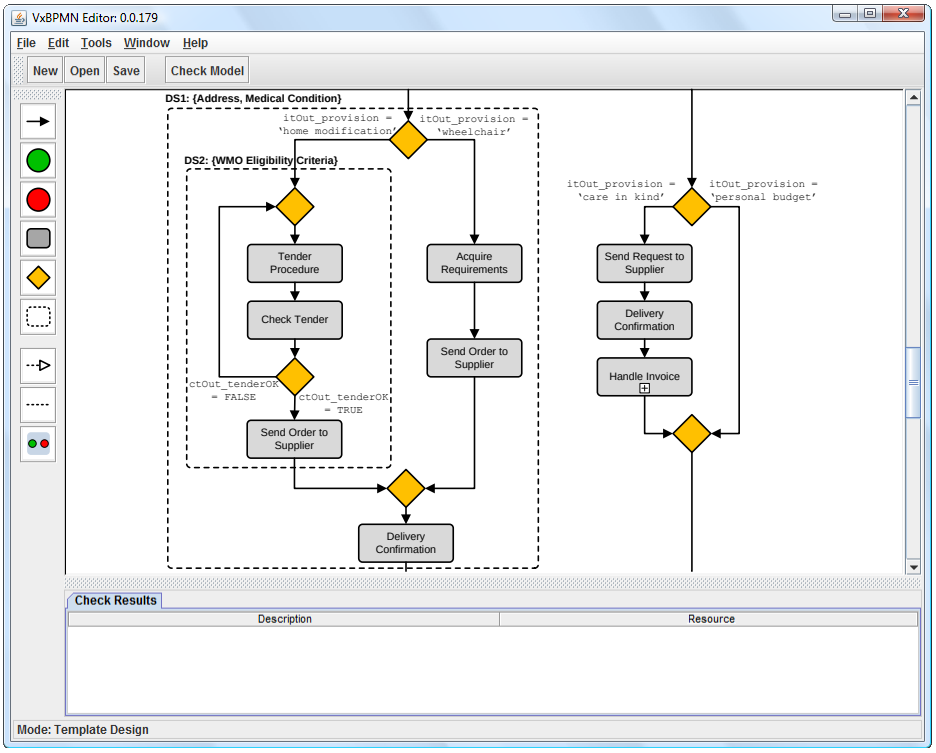


Figure 5.4: *Screenshot of the Process Modeller.*

Figure 5.4 presents a screenshot of the PM, showing the graphical representation of DS1 and DS2 of the WMO process from Figure 5.1. The DSs are saved along with the process specification itself. The final output of the PM is an XML representation of the BP, which conforms to Definition 5.2. This representation is passed to the PE for execution, as described in the next subsection.

5.5.2 The Process Executor

The Process Executor (PE) is responsible for executing a BP as specified by the PM. The PE takes as input a BP specification in conformance with an XML schema that represents Definition 5.2, and with the BP input parameters instantiated to specific values. The PE works in cooperation with the Service Repository as described in Definition 5.1. The details of Service Instances implementation are out of scope of this study, and for the purposes of the testing presented in Section 5.6 the service invocations are simulated.

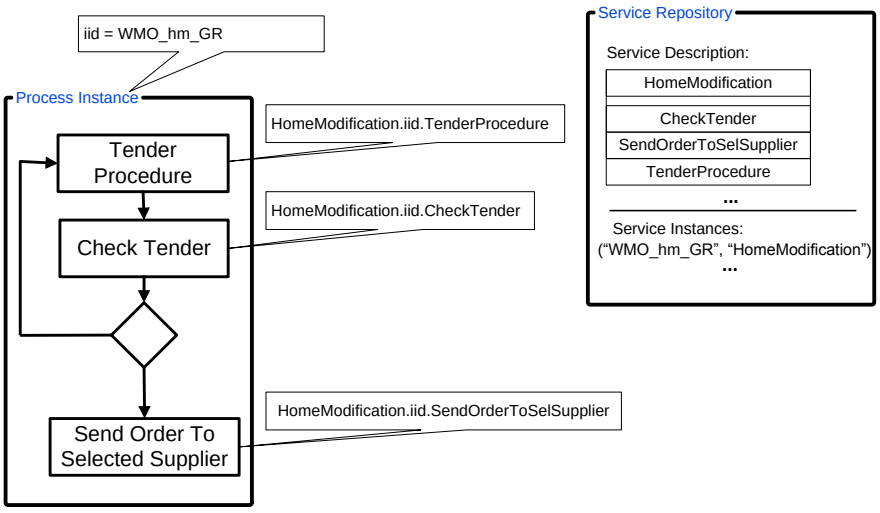


Figure 5.5: *Example of a Service Description and a Service Instance.*

The activities included in the BP specification must refer to method invocations that can be retrieved from the SR. Given a fully qualified reference to an invocation method *sdid.iid.oid* specified by an activity in the BP specification, the PE retrieves the respective description kept in the SR. For example, the activity "*Send Order*" in Figure 5.5 refers to "*HomeModification.iid.sendOrderToSel-Supplier*," which corresponds to the method "*sendOrderToSelSupplier*" of the "*HomeModification*" service description, and is provided by the service instance with identifier "*WMO_hm_GR*" (see Definition 5.1). The service description of "*HomeModification*" as well as the service instance (provider) "WMO_hm_GR" are kept in the SR, as shown in Figure 5.5. It should be noted that the value of the variable iid in the BP specification may be unknown before a process is actually started, and an assignment to another value $iid = iv$ can be used instead of a predefined value. The value of iv can be provided by the user at execution time, or retrieved by the PE as an output value of a service

method call. In the example in Figure 5.5 the value "WMO_hm_GR" for the variable iid is provided at the time the process instance execution starts.

In the current implementation, an activity is executed by directly invoking the respective method, without checking whether the preconditions prescribed in the corresponding service instance description in the AA hold. The reason for such a simplification is to make the process executer to operate in the same way as a typical execution engine, thus providing backward compatibility.

The data flow and knowledge about the environment are handled by a *local storage* (LS), which is maintained by the PE and reflects its knowledge about the environment and the state of the process instance execution. Some of these variables are specific to a particular BP running instance, and some are common to multiple BPs. During execution, the PE updates the LS according to the new information it receives from the environment (from service method invocations), and to the specifications included in the BP description (assignments to variables). When the PE receives a request for executing an instance of a BP specification $BP = (PV, E)$, it assigns a unique identifier $bp\text{-}iid$ to the running instance, and constructs the AA along with the instance-level inputs and outputs $AI \cup AO$ (as described in Section 5.4.1), which are added to the LS. Each service state variable $sv \in ASV$ (see Section 5.4.1) is added to the LS if it does not already exist. This way, state variables of the AA are shared among running process instances, whereas instance-level input and output variables are unique to each process instance. Moreover, the PE constructs the instance-level internal variables declared in the BP (i.e. for each $var \in PV_i$) with name v a variable with name $bp\text{-}iid.v$ and domain identical to var's domain is added to the LS. The internal process variables are also unique to the process instance. The value of an instance-level variable cannot be changed by any other external factor other than the BP instance $bp\text{-}iid$ it belongs to, while a shared variable can be modified by any other entity that calls the service operation which affects it.

The distinguishing feature of the PE with respect to other well-known BP execution engines is the support for dealing with the DSs specified in a BP. When a process execution runs into a DS, the PE turns into a special "DS mode." In that mode, the PE creates an event listener for each of the volatile variables specified in the DS. It is assumed that modification events can be captured by subscribing to specific variables of interest, and that external services that have the permission to change these variables, publish an appropriate event that is caught by the subscribed clients (listeners). The details of event firing and catching are out of scope of the study.

The event handling is deferred until the activity currently being executed finishes, thus avoiding potential inconsistencies that may result from canceling an activity in the middle of execution. Therefore, the information conveyed by the data modification events is stored in a memory list that maintains tuples of the re-

cently modified variables and their latest values. A new event on the same variable overwrites the old value of the variable kept in the memory list. This list of recent changes is checked prior to executing the next activity within a DS, and if it is not empty, the conditions in the *verify* block of the DS are checked towards the latest values kept in the list. If a condition evaluates to true, the respective goal or process element is fired, while the BP execution is suspended. In case of a flow, all parallel branches are put on hold. The list of recent changes is cleared, and the LS is updated accordingly, by incorporating the most up-to-date values to the respective variables.

In case a goal has to be pursued, the planner is invoked in order to create a plan which is then executed, while in the case of a pre-specified element this is directly executed. After a plan or a pre-specified element is executed the initial process execution is resumed, starting from the activity which is immediately after the end of the current DS. In case parallel branches were suspended, these are resumed as well (the underlying assumption is that the execution of the generated IP does not introduce any inconsistencies in the suspended concurrent branches). The only exception is when there is a *terminate* annotation referring to the goal that is triggered (see Definition 5.3), in which case the original BP is terminated instead.

In case of nested DSs, the conditions are verified for all active dependency scopes starting from the most outer one and going inward. When the execution of a subprocess covered by some DS is finished, then the respective DS is removed from the list of active DSs, as well as all event listeners associated with it. If the list is empty, then PE leaves the "DS mode" and does not listen to any data modification events. Note that while executing an IP, the PE still remains in the same "DS mode," and thus treats the modification events it receives during the IP execution in the same way as it did during execution of the process element covered by the DS in the BP. This means that an IP "inherits" the DSs that covered the activity before which the planner was invoked. In case a DS condition is triggered, the current IP execution is interrupted, a new IP is generated, after whose completion, the PE returns to the state after the DS in the original BP.

In order to generate a plan, the AI planner needs a planning domain representation (see Definition 5.4). To this end, the PE calls the Domain Generator, by passing to it the Atomic Actions (AA), built as described in Section 5.4.1 by including all service instances referenced in the BP and a set of eligible compensation services from the SR. The planning domain is constructed only once for a specific BP, the first time that a DS is triggered. The goal taken from the DS specification and the current state, i.e. the values of the variables that are part of the planning domain as reflected by the updated database, are handed over to the AI planner, which uses them along with the planning domain to compute a plan. This plan, which includes only sequence and flow structures, is then passed for execution to the PE. Loops in

the plan are "flattened," i.e. the plans explicitly include all repetitions in sequence. Deferred choices (such as in the case of XORs) are addressed indirectly as already described in Section 5.4.3: whenever the PE executes an operation that returns a new value, the constraint solver is called to check whether this value leads to any inconsistencies with respect to the outcome anticipated by the plan, and if so, the planner is re-invoked with the current state of execution as the initial state (and the same goal).

5.5.3 The planner

The planner is implemented in Java, and communicates with the PE through standard method calls. Upon receiving a request for computing a plan from the PE, the planner translates the BP-specific planning domain, the initial state and the goal it received into a CSP, as presented in Section 5.3.3. A standard constraint solver is applied to solve the CSP, in order to find a solution that amounts to a valid plan. The Choco v2.1.1 constraint programming library [1] is used, which provides a large choice of implemented constraints, as well as a variety of pre-defined and custom search methods. The solution to a CSP amounts to a partially ordered plan, i.e. one that may contain parallel actions if not restricted by interdependencies between actions. This plan is passed to the PE for execution, as described in the previous section.

5.6 Evaluation

The aim of the evaluation is (i) to demonstrate the effectiveness of our approach with respect to our working example presented in Section 5.5 and (ii) to test the performance with respect to the time that is required to generate the necessary IPs. The specification of the desired goals and DSs has been conducted in close cooperation with WMO employees at the municipality of Groningen. Our experience confirmed that the translation of the requirements as expressed by non-technical employees to the representation required by our framework is rather intuitive, and is relatively easily understood when shown to non-experts for proof-checking.

In the tests presented in the next subsection, service invocations are simulated, and the methods provided by the service instances have a predefined behaviour, simulated according to the different situations we want to test. The performance of the framework has been tested with respect to atomic action repositories of increasing size, since domains that comprise a large set of actions, may raise concerns of

[1] `www.emn.fr/z-info/choco-solver`

inefficiency.

For the case of the WMO process taken from the case study (Section 1.1.1, Figure 1.4), the experiments on generating of intervention processes displayed good performance results. For the plans comprising of 5 to 7 activities, the time required to calculate the plan lies within one second.

In addition to the tests on the real business process, we performed scalability tests on a virtual test set of 100 atomic actions. The tests demonstrated that for a trivial domain, less than 6 sec are required to generate an IP comprising as many as 30 activities. These results confirm that the time for generating an IP in realistic situations is a matter of a few seconds, which is an acceptable performance considering the average throughput time of long-running BPs (varying between 1 and 6 weeks for the WMO case). The detailed description of the tests together with their results can be found in [van Beest et al. 2012].

5.7 Discussion

In this chapter, an approach is presented for automated runtime process reconfiguration, which ensures the recovery of a BP from erroneous states without the necessity of predefining all potential interference situations, and the respective ways to overcome them. We have studied the feasibility of our approach with respect to a real scenario, the business process of the Dutch WMO law. We show how AI planning can be used to ensure the consistency of the process execution results in an automatic way, given a number of high-level annotations in terms of dependency scopes provided by the domain designer. The application of the approach in business domains where data can be changed by external factors, can highly benefit organizations by resolving potential inconsistencies in a way that enables a higher degree of flexibility by reducing hard-coded dependency solutions and workflow repair mechanisms.

To evaluate the feasibility of the approach, an architecture has been designed and a prototype has been implemented. The results indicate that coupling DSs with declarative goals and generating IPs at runtime by means of AI planning is a usable and realistic method for resolving inconsistencies caused by process interference. The proposed method is both sound and complete. That is, the generated IPs always satisfy the properties specified in the goal, and if there exists a combination of activities that achieves the goal, then this sequence is found.

The work presented in this chapter has the affinity with the problem of transation management in business processes [Grefen, Vonk and Apers 2001]. The situation with transactions, however, is very specific in a way that it is important to recover

the process and compensate the negative effects. This work, on the other hand, is mainly focused on business process reconfiguration as a way to address the changes in the underlying requirements or the modification in the execution context, when the task of consistency keeping is not of the main concern.

Although the focus of the current study is to deal with inconsistencies that result from process interference, the overall approach based on domain-independent AI planning for BP reconfiguration is more general. For example, the system can be extended so that it can be used for process adaptation in case of changes in the business requirements/rules. The dynamic nature of the CSP on which the planning framework is based on allows the incorporation of changes in the BP-specific constraints at runtime: constraints which become obsolete can be removed on-the-fly from the constraint network, and the same holds for the addition of new constraints.

Our precondition and effects language is similar in spirit with existing semantic markups for Web Services such as OWL-S. Finding a suitable and yet powerful interface for designing goals and service descriptions and integration with existing standards is open for future investigation.

According to the approach presented herein, a recovery process is only fired in case of change events that are covered by dependency scopes. However, there may be environmental changes that compromise the consistency of the Business Process, and have not taken into consideration by the domain designer. In order to prevent potential erroneous situations resulting from such events that have not been anticipated at design time, we also consider to adopt a conservative policy that holds the execution in case a violation with respect to the BP specification is detected. This can be done by extending the process executor, so that it can check whether the preconditions of an action, as specified in the semantic repository, indeed hold at execution time. This will imply some extra cost for violation checks, and is similar to the approach for mismatch detection presented in [Marrella and Mecella 2011]. These extra checks for violations during execution will also help address cases where the IP may lead to inconsistencies with respect to possible concurrent activities that were put to hold during the recovery process.

The work presented herein shares many concerns with different subfields of BP management, including work in the areas of BP recovery, adaptation and process interference. Automated planning for the purpose of runtime BP reconfiguration has been proposed earlier in the literature.

Although adaptation of processes to resolve process interference can be considered a very specific form of changeability, existing changeability frameworks are primarily requirements-driven. That is, their adaptation capabilities are specially tailored to facilitate and support new business requirements (and, therefore, improve flexibility), whereas they do not incorporate the mechanisms to adapt the process in

order to prevent erroneous business outcomes. Consequently, requirements-driven changeability and adaptability can be considered orthogonal to our research.

The methodology adopted herein shares many common concerns with the work on semantic service composition by adopting AI planning techniques [Sohrabi and McIlraith 2010, Kaldeli et al. 2011, Au et al. 2005], since the ultimate task is to combine actions in a dynamic way. The common premise underlying these approaches is that services come along with semantic annotations that describe their behaviour in some convenient format, usually in terms of preconditions and effects, which is the representation we follow in the current work as well. Many of the approaches proposed for service composition via automated planning, however, require that the set of supported solutions is pre-defined in some form of procedural templates, like in [Sohrabi and McIlraith 2010, Au et al. 2005]. On the contrary, our approach of ad-hoc automatic process instance reconfiguration relies on a domain-independent planner that has been first presented in [Kaldeli et al. 2009], where the domain designer just states *what* properties have to be satisfied, without having to anticipate *how* these can be fulfilled.

Temporal aspects, maintainability properties, and the distinction between the wish to observe the environment or change it are some of the features this language supports [Kaldeli et al. 2009]. Moreover, due to the internal reformulation of the problem as a Constraint Satisfaction Problem (CSP), the planner naturally supports efficient handling of variables ranging over large domains, which are commonly used by BPs (dates, locations etc). Furthermore, the planner is equipped with a knowledge-level representation to deal with the problem of incomplete knowledge and sensing, which is particularly well-suited in the case of XORs evaluating a condition, whose output is unknown off-line. This way, plans are automatically built based on the agent's knowledge and the way that this is changed by actions. Replanning is employed to deal with the large number of possible outcomes and unforeseen contingencies at runtime, as described in [Kaldeli et al. 2011].

Chapter 6

Conclusion

In this thesis we proposed a framework for operating of multiple variants of the same business process. One of the prominent features of our framework is the handling of variations which may happen at the time of business process modeling as well as the variations which may happen at runtime. Additionally, we cover the problem of business process migration, when a change in a generic business process has to be propagated to each of the individual customized variants, and, additionally, possible modification of a generic process can be verified against existing variants.

The framework is based on the application of the formalisms of propositional and temporal logic to describe a business process model as a set of logical formulas. The first advantage of such a representation is the ability to describe a whole family of business processes at once via simple manipulations with logical formulas. We provide an intuitive way to describe a business process or a family of business process by the means of a visual modeling language, which is based on BPMN notation enhanced with additional modeling elements. These additional elements range from simple unidirectional arrows reflecting some relationship between two business process elements to complicated group elements embracing whole sub–processes. For each of the visual elements there is a transformation rule which prescribes how to convert that element into a set of CTL formulas.

Comparing with the state of the art design–time frameworks we emphasize that our framework offers both declarative and imperative variability techniques. Because of its declarative foundation, it inherits the expressivity provided by that kind of frameworks [van der Aalst and Pesic 2006, Pesic et al. 2007]. At the same time, declarative usability issues are ameliorated through a straightforward to use visual modeling extension from which the declarative constraint formulas can be generated directly.

The second advantage of our solution is the ability to analyze the difference between two or more business process models and describe that difference in the form of a transformation function. Such a way to represent a business process makes it possible to analyze the difference between any two given business processes as a result of matrix transformation. We analyze the properties of the matrices which rep-

resent business processes, and can tell the correspondence between manipulation with such matrices and the respective changes in a business process model inflicted by such manipulation; and we distinguish between adding, removing, moving, and swapping of the activities in a business process model. These four manipulations we call **primitive**, and they all possess the same important property that they can be applied to any business process and the result of such a manipulation is also a business process.

We proved that property via analyzing the basic properties of the matrices that represent business processes, and noticing that any of the aforementioned primitive manipulations over a matrix retains those basic properties. A transformation from one business process model to another can be easily extracted working with process matrices only, since such a transformation can be represented as an algebraic operation over matrices. As a result, we reduce the task of extraction of such transformation functions to the task of identifying a proper **permutations** of rows and columns of a process matrix, that is, such a replacement of row and columns in the starting matrix so that the transformed one is the desired matrix.

If such a permutation does not exist, then we look instead for a permutation which makes the starting matrix as close as possible to the desired matrix. The distance between any two matrices in our case is the number of their non–equivalent cells, and it tells us what is the degree of mistake when we are approximating a real matrix transformation with a primitive one (the one which is a combination of primitive transformation).

The ideas behind annotation–based variability are in line with the task of business process transformation, however, the possible modifications should be known in advance and introduced in the form of annotations or explicit variability points. On the contrary, the ideas of business process transformation presented in this thesis are more general and do not require any prior analysis.

The third advantage of our framework is the ability to execute a business process in such a way that the actual sequence of actions is decided at runtime on the basis of the context of execution. We also implemented a special business process orchestration engine, which takes as input a business process described in BPEL–like language. The distinctive features of our engine comparing with existing ones are (i) it listens to the data modification events, and, (ii), if necessary, stops the execution of the business process and starts a so–called *repair process*. Such a repair process could contain the activities of the alternative branch of the business process together with the activities which compensate for the result of the activities which were already executed.

The first feature is achieved with the help of so–called dependency scopes, which are essentially sub–processes which are dependent on certain data variables. If one

of such variables attached to a dependency scope is modified while the execution of a business process is within the dependency scope, then the execution of a business process is suspended, and a repair process is created.

The creation of a repair process is up to an AI planner, which is actually a domain-independent planner based on dynamic Constraint Satisfaction Problem (CSP) techniques [Kaldeli et al. 2009]. The planner takes as input a declarative goal which is attached to the corresponding dependency scope and a planning domain which contains all the variables relevant to the business process together with their actual values. The generated repair processes always satisfy the properties specified in the goal, and if there exists a combination of activities that achieves the goal, then this sequence is found and returned back to the orchestration engine for later execution.

To summarize, in this thesis we proposed a framework which supports business process variability at both design– and run– time with additional support for business process evolution. The idea, however, is the same for all of those facets of business process management and is based on the representation of a business process as a set of logical formulas, which are then processed and analyzed with the help of existing tools and methodologies. Our framework was also verified on real–life business processes taken from the field of Dutch e–Government.

There are several possible extensions to the research conducted within the scope of this thesis. First, the visual modeling language we developed to support the design–time variability can be enhanced to support an automated creation of business process variants. Currently, we support the automated creation at runtime which is driven by the execution context, and the result business process is actually a sequence of activities which covers one particular situation only. By enriching our visual modeling language with additional annotations, we can generate a whole business process model derived from the original template based on specific requirements provided by a user.

Second, our language for business process representation can be enriched to distinguish not only different kind of branching points (OR– and AND–), but also to take into account the logical conditions which are typically assigned to different branches of an OR– or XOR– gateway. This will also solve the issue of not supporting of XOR–gateways explicitly, and expand our language to cover both behavioral and data–flow models of a business process.

And finally, our framework can be upgraded to a business process repository, that is, in addition to covering the issues related to variability, the framework should also provide additional features. Those features include the ability to store a collection of business processes together with their variants, keeping track of the changes in business processes, providing with search for and comparison of business process model as provided by the state of the art BP repositories [Dijkman et al. 2012].

Appendix A

Modal Logics in a Nutshell

Modal propositional logic is a formal logic which can tell about the truth of propositions with regard to other conditions. Formally, the **syntax** of modal logic is defined recursively as follows [Blackburn et al. 2001]:

1. Each propositional formula is a modal formula;
2. If p is a modal formula, so is $\neg p$;
3. If p, q are modal formulas, so is $p \wedge q$;
4. If p is a modal formula, so is $\Box p$.

Additionally, a special symbol $\Diamond$ is used in order to simplify the expressions of modal logic, and the meaning of this symbol is the following: $\Diamond p \equiv \neg\Box\neg p$, where p is a modal formula.

A **model** is needed in order to tell the truth about a given modal formula. In other words, the same formula can be valuated to TRUE ($\top$) of FALSE ($\bot$) depending on the **model** under consideration. In order to define a model, one needs to define a **frame** and a **valuation function**.

Kripke structure [Blackburn et al. 2001] is used as a frame for modal logic, and this structure is defined as a tuple $\mathcal{K} = \langle S, R \rangle$, where

1. S is a non–empty set of states, and
2. $R \subseteq S \times S$ is a set or relations.

In other words, a Kripke structure can be seen as a directed graph. A **valuation function** w.r.t. a given Kripke structure $\mathcal{K} = \langle S, R \rangle$ is a function $L : S \to 2^{AP}$, where AP is a set of atomic propositions. In other words, a valuation function tells which propositions at which states are equal to TRUE.

Given a model $\mathcal{M}$, which in turn consists of a Kripke structure $\mathcal{K} = \langle S, R \rangle$ and a valuation function L, the truth of a modal formula can be defined w.r.t. a given state $x \in S$. If a formula p is valuated to $\top$ at a state x of a model $\mathcal{M}$ it is formally written as $\mathcal{M}, x \models p$. The truth of a given modal formula is defined recursively as follows:

1. $\mathcal{M}, x \models a \Leftrightarrow a \in L(x)$ for any atomic proposition a;
2. $\mathcal{M}, x \models \neg p \Leftrightarrow$ not $\mathcal{M}, x \models p$;
3. $\mathcal{M}, x \models p \wedge q \Leftrightarrow \mathcal{M}, x \models p \wedge \mathcal{M}, x \models q$;
4. $\mathcal{M}, x \models \Box p \Leftrightarrow \forall y : (x, y) \in R : \mathcal{M}, y \models p$.

There are many different modal logics, but in the scope of this thesis we are mostly interested with Linear Temporal Logic (LTL) and Computational Tree Logics (CTL, CTL$^+$, CTL*).

A.1 Linear Temporal Logic (LTL)

Linear temporal logic [Pnueli 1977] is a modal propositional logic with its modality related to time. The syntax of LTL is defined recursively as follows:

1. Each propositional formula is an LTL formula;
2. If p, q are LTL formulas, then so are $(p \wedge q)$ and $\neg p$;
3. If p is an LTL formula, then $\circ p$, $\Diamond p$ and $\Box p$ are also LTL formulas;
4. If p, q are LTL formulas, then $[pUq]$ and $[pWq]$ are also LTL formulas.

The formulas of LTL are evaluated for a given model $\mathcal{M}$ for a given path π. A path $\pi = \{x_1, x_2, \ldots x_n, \ldots\}$ is a potentially infinite sequence of states $x_i \in S$, when $\forall i : x_i, x_{i+1} \in \pi \Leftrightarrow (x_i, x_{i+1}) \in R$, where $\mathcal{K} = \langle S, R \rangle$ is a frame of the model $\mathcal{M}$.

The truth of LTL formulas is defined recursively as follows for a given path $\pi = \{x_1, x_2, \ldots\}$:

1. $\mathcal{M}, \pi \models a \Leftrightarrow a \in L(x_1)$ for any atomic proposition a;
2. $\mathcal{M}, \pi \models \neg p \Leftrightarrow$ not $\mathcal{M}, \pi \models p$;
3. $\mathcal{M}, \pi \models p \wedge q \Leftrightarrow \mathcal{M}, \pi \models p \wedge \mathcal{M}, \pi \models q$;
4. $\mathcal{M}, \pi \models \circ p$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \mathcal{M}, x_2 \models p$;
5. $\mathcal{M}, \pi \models \Diamond p$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \exists x_i \in \pi : \mathcal{M}, x_i \models p$;
6. $\mathcal{M}, \pi \models \Box p$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \forall x_i \in \pi : \mathcal{M}, x_i \models p$;
7. $\mathcal{M}, \pi \models pUq$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \exists n : \forall i = 1 \ldots (n-1) : \mathcal{M}, x_i \models p \wedge \mathcal{M}, x_n \models q$;
8. $\mathcal{M}, \pi \models pWq \Leftrightarrow \mathcal{M}, \pi \models pUq \vee \mathcal{M}, \pi \models Gp$.

A.2 Computational Tree Logic (CTL)

Computational Tree Logic (CTL) and Computational Tree Logic$^+$ (CTL$^+$) are Branching–time Logics and are defined recursively as follows by applying rules 1-5 and 1-6 respectively [Clarke et al. 1986, Emerson and Halpern 1985].

1. Each primitive formula is a state formula;
2. If p, q are state formulas, then so are $(p \wedge q)$ and $\neg p$;
3. If p is a state formula, then Xp, Fp and Gp are path formulas;
4. If p is a path formula, then Ep and Ap are state formulas;
5. If p, q are state formulas, then $[pUq]$ and $[pWq]$ are path formulas;
6. If p, q are path formulas, then so are $p \wedge q$ and $\neg p$.

The semantics of CTL is more complicated than the semantics of LTL because of the distinction between path and state formulas. A path p is a potentially infinite sequence $\pi = \{x_1, x_2, \ldots x_n, \ldots\}$, defined in the same way as it was defined for LTL. The truth of CTL and CTL^+ is defined recursively as follows, when a state formula is evaluated at a state x, and a path formula is evaluated at a path π:

1. $\mathcal{M}, x \models a \Leftrightarrow a \in L(x)$ for any atomic proposition a;
2. $\mathcal{M}, x \models \neg p \Leftrightarrow$ not $\mathcal{M}, x \models p$;
3. $\mathcal{M}, x \models p \wedge q \Leftrightarrow \mathcal{M}, x \models p \wedge \mathcal{M}, x \models q$;
4. $\mathcal{M}, x \models Ap$ means that for all paths p_i in the model $\mathcal{M}$ which start in the state x, the path formula p holds for those paths;
5. $\mathcal{M}, x \models Ep$ means that there is a path p_i in the model $\mathcal{M}$ which start in the state x, the path formula p holds for the path p_i;
6. $\mathcal{M}, \pi \models Xp$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \mathcal{M}, x_2 \models p$;
7. $\mathcal{M}, \pi \models Fp$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \exists x_i \in \pi : \mathcal{M}, x_i \models p$;
8. $\mathcal{M}, \pi \models Gp$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \forall x_i \in \pi : \mathcal{M}, x_i \models p$;
9. $\mathcal{M}, \pi \models pUq$ means that for the path $\pi = \{x_1, x_2, \ldots\} : \exists n : \forall i = 1 \ldots (n-1) : \mathcal{M}, x_i \models p \wedge \mathcal{M}, x_n \models q$;
10. $\mathcal{M}, \pi \models pWq \Leftrightarrow \mathcal{M}, \pi \models pUq \vee \mathcal{M}, \pi \models Gp$;

11. $\mathcal{M}, \pi \models \neg p$ means that not $\mathcal{M}, \pi \models p$;

12. $\mathcal{M}, \pi \models p \wedge q \Leftrightarrow \mathcal{M}, \pi \models p \wedge \mathcal{M}, \pi \models q$.

The rules 1–10 are relevant for both CTL and CTL^+, and the rest of the rules are relevant for CTL^+ only.

Bibliography

Adams, M., ter Hofstede, A., Edmond, D. and van der Aalst, W.: 2006, Worklets: A service-oriented implementation of dynamic flexibility in workflows, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pp. 291–308.

Aiello, M., Bulanov, P. and Groefsema, H.: 2010, Requirements and tools for variability management, *IEEE workshop on Requirement Engineering for Services (REFS 2010) at IEEE COMPSAC*, pp. 245 – 250.

Allen, J. F.: 1983, Maintaining knowledge about temporal intervals, *Commun. ACM* **26**(11), 832–843.

Au, T., Kuter, U. and Nau, D.: 2005, Web Service Composition with Volatile Information, *Prof. of the 4th Int. Semantic Web Conf. (ISWC)*.

Augustus, M. A. A.: 167 A.D., *Meditations*, Vol. 10 of *Meditations*, Handwritten.

Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B. and Vilbig, A.: 2004, A meta-model for representing variability in product family development, *Software Product-Family Engineering*, Vol. 3014, pp. 66–80.

Balko, S., ter Hofstede, A. H. M., Barros, A. P. and La Rosa, M.: 2009, Controlled flexibility and lifecycle management of business processes through extensibility, *EMISA*, pp. 97–110.

Basten, T. and van der Aalst, W. M.: 2001, Inheritance of behavior, *Journal of Logic and Algebraic Programming* **47**(2), 47 – 145.

Becker, J., Delfmann, P. and Knackstedt, R.: 2007, Adaptive reference modeling: Integrating configurative and generic adaptation techniques for information models, *Reference Modeling*, pp. 27–58.

Beckstein, C. and Klausner, J.: 1999, A meta level architecture for workflow management, *Journal of Integrated Design and Process Science* **3**, 15–26.

Blackburn, P., Rijke, M. D. and Venema, Y.: 2001, *Modal Logic*, Cambrige University Press.

Brogi, A. and Popescu, R.: 2006, Towards semi-automated workflow-based aggregation of web services, *CIbSE*, pp. 9–22.

Bulanov, P., Lazovik, A. and Aiello, M.: 2011, Business process customization using process merging techniques, *In Int. Conf. on Service-Oriented Computing and Applications*, pp. 1–4.

Charfi, A. and Mezini, M.: 2004, Hybrid web service composition: business processes meet business rules, *ICSOC*, pp. 30–38.

Clarke, E. M., Emerson, E. A. and Sistla, A. P.: 1986, Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263.

Clarke, E. M., Grumberg, O. and Peled, D. A.: 2000, *Model Checking*, MIT Press.

Cleaveland, R.: 1999, Temporal process logic, *CONCUR99 Concurrency Theory*, Vol. 1664, pp. 779–779.

Clements, P.: 2006, Managing variability for software product lines: Working with variability mechanisms, *Software Product Line Conference, 2006 10th International*, pp. 207–208.

Cook, S. A.: 1971, The complexity of theorem-proving procedures, *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pp. 151–158.

Czarnecki, K. and Eisenecker, U.: 2000, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

Dadam, P. and Reichert, M.: 2009, The ADEPT project: a decade of research and development for robust and flexible process support, *Computer Science - R&D* **23**(2), 81–97.

de Leoni, M., De Giacomo, G., Lespèrance, Y. and Mecella, M.: 2009, On-line adaptation of sequential mobile processes running concurrently, *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC'09, ACM, pp. 1345–1352.

de Leoni, M., Mecella, M. and De Giacomo, G.: 2007, Highly dynamic adaptation in process management systems through execution monitoring, *BPM 2007* pp. 182–197.

Delfmann, P. and Knackstedt, R.: 2007, Towards tool support for information model variant management - a design science approach, *ECIS*, pp. 2098–2109.

Dijkman, R., La Rosa, M. and Reijers, H. A.: 2012, Managing large collections of business process modelscurrent techniques and challenges, *Computers in Industry* **63**(2), 91 – 97.

Dijkman, R. M., Dumas, M., van Dongen, B. F., Käärik, R. and Mendling, J.: 2011, Similarity of business process models: Metrics and evaluation, *Inf. Syst.* **36**(2), 498–516.

Emerson, E. A. and Halpern, J. Y.: 1985, Decision procedures and expressiveness in the temporal logic of branching time, *Journal of Computer and System Sciences* **30**(1), 1–24.

Eshuis, R. and Grefen, P. W. P. J.: 2009, Composing services into structured processes, *International Journal of Cooperative Information Systems* **18**(2), 309–337.

Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H. and Wolf, K.: 2009, Instantaneous soundness checking of industrial business process models, *7th Int. Conf. on Business Process Management*, pp. 278–293.

Ferreira, H. and Ferreira, D.: 2006, An integrated life cycle for workflow management based on learning and planning, *International Journal of Cooperative Information Systems* **15**, 485–505.

Gajewski, M., Meyer, H., Momotko, M., Schuschel, H. and Weske, M.: 2005, Dynamic failure recovery of generated workflows, *Database and Expert Systems Applications (DEXA) Workshops*, IEEE Computer Society Press, pp. 982–986.

Galvão, I., van den Broek, P. and Akşit, M.: 2010, A model for variability design rationale in spl, *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pp. 332–335.

Garcia-Molina, H. and Salem, K.: 1987, Sagas, *Proc. of 1987 ACM SIGMOD Int. Conf. on Management of data*, ACM, pp. 249–259.

Ghallab, M., Nau, D. and Traverso, P.: 2004, *Automated Planning: Theory and Practice*, Elsevier. Chapters 13,14.

Giannakopoulou, D. and Havelund, K.: 2001, Automata-based verification of temporal properties on running programs, *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pp. 412–416.

Gottschalk, F., van der Aalst, W. M. P., Jansen-Vullers, M. H. and La Rosa, M.: 2008, Configurable workflow models, *Int. J. Cooperative Inf. Syst.* **17**(2), 177–221.

Grefen, P., Vonk, J. and Apers, P.: 2001, Global transaction support for workflow management systems: from formal specification to practical implementation, *The VLDB Journal* **10**, 316–333.

Groefsema, H., Bulanov, P. and Aiello, M.: 2011, Declarative enhancement framework for business processes, *Int. Conference on Service-Oriented Computing, ICSOC, LNCS 7084*, pp. 495–504.

Hallerbach, A., Bauer, T. and Reichert, M.: 2008, Managing process variants in the process life cycle, *ICEIS (3-2)*, pp. 154–161.

Hoffmann, J., Weber, I. and Kraft, F.: 2010, SAP Speaks PDDL, *4th National Conf. of the American Association for Artificial Intelligence (AAAI'10)*.

Jablonski, S.: 1994, MOBILE - a modular workflow model and architecture, *4th Int. Conf. on Dynamic Modelling and Information Systems*.

Jarvis, P., Moore, J., Stader, J., Macintosh, A., Casson-du Mont, A. and Chung, P.: 1999, Exploiting ai technologies to realise adaptive workflow systems, *In Proc. AAAI Workshop on Agent-Based Systems in the Business Context, 1999*, AAAI Technical Report WS-99-02.

Kaldeli, E., Lazovik, A. and Aiello, M.: 2009, Extended goals for composing services, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, AAAI Press.

Kaldeli, E., Lazovik, A. and Aiello, M.: 2011, Continual planning with sensing for Web Service composition, *Proc. of the 25th AAAI Conf. on Artificial Intelligence (to appear)*, AAAI Press.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. and Peterson, A. S.: 1990, Feature-oriented domain analysis (FODA) feasibility study, *Technical Report CMU/SEI-90-TR-21*, Carnegie-Mellon University, Software Engineering Institute.

Kang, K., Lee, J. and Donohoe, P.: 2002, Feature-oriented product line engineering, *Software, IEEE* **19**(4), 58 – 65.

Kopp, O., Martin, D., Wutke, D. and Leymann, F.: 2008, On the choice between graph-based and block-structured business process modeling languages, *Modellierung betrieblicher Informationssysteme (MobIS 2008)*, Vol. 141 of *Lecture Notes in Informatics (LNI)*, pp. 59–72.

Küster, J. M., Gerth, C., Förster, A. and Engels, G.: 2008, A tool for process merging in business-driven development, *CAiSE Forum*, pp. 89–92.

La Rosa, M., Dumas, M., Uba, R. and Dijkman, R.: 2010, Merging business process models, *OTM Conferences (1)*, pp. 96–113.

La Rosa, M., Lux, J., Seidel, S., Dumas, M. and ter Hofstede, A.: 2007, Questionnaire-driven configuration of reference process models, *Advanced Information Systems Engineering*, Vol. 4495, pp. 424–438.

La Rosa, M., van der Aalst, W., Dumas, M. and ter Hofstede, A.: 2009, Questionnaire-based variability modeling for system configuration, *Software and Systems Modeling* **8**, 251–274.

Lapouchnian, A., Yu, Y. and Mylopoulos, J.: 2007, Requirements-driven design and configuration management of business processes, *BPM*, pp. 246–261.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F. and Scherl, R. B.: 1997, GOLOG: A logic programming language for dynamic domains, *The Journal of Logic Programming* **31**(13), 59 – 83.

Liang, Q., Chakarapani, L. N., Su, S. Y. W., Chikkamagalur, R. N. and Lam, H.: 2004, A semi-automatic approach to composite web services discovery, description and invocation, *Int. Journal Web Service Research* **1**(4), 64–89.

Lu, R., Sadiq, S. and Governatori, G.: 2009, On managing business processes variants, *Data Knowl. Eng.* **68**(7), 642–664.

Madhusudan, T., Zhao, J. L. and Marshall, B.: 2004, A case-based reasoning framework for workflow model management, *Data Knowl. Eng.* **50**, 87–115.

Marrella, A. and Mecella, M.: 2011, Continuous planning for solving business process adaptivity, *12th International Working Conference on Business Process Modeling, Development and Support (BPMDS 2011), in conjunction with CAiSE 2011*.

Milner, R.: 1989, *Communication and concurrency*.

Milner, R.: 1999, *Communicating and Mobile Systems: The Pi Calculus*, Cambridge University Press.

Momotko, M. and Subieta, K.: 2004, Process query language: A way to make workflow processes more flexible, *ADBIS*, pp. 306–321.

Müller, R., Greiner, U. and Rahm, E.: 2004, AgentWork: a workflow system supporting rule-based workflow adaptation, *Data & Knowledge Engineering* **51**(2), 223 – 256.

Murata, T.: 1989, Petri nets: Properties, analysis and applications, *Proceedings of the IEEE* **77**(4), 541 –580.

Object Management Group (OMG): 2009, Business process model and notation (BPMN) FTF beta 1 for version 2.0, *Technical Report dtc/2009-08-14*, OMG. http://www.omg.org/spec/BPMN/2.0, accessible July 2012.

(OMG), O. M. G.: 2005, UML 2.0 unified modeling language, *Technical Report formal/2005-07-05*, OMG. http://www.omg.org/spec/UML/2.0, accessible July 2012.

Ouvans, C., Dumas, M., ter Hofstede, A. and van der Aalst, W.: 2006, From bpmn process models to bpel web services, *Web Services, 2006. ICWS '06. International Conference on*, pp. 285–292.

Pearl, J.: 1984, *Heuristics: intelligent search strategies for computer problem solving*, Addison-Wesley Longman Publishing Co., Inc.

Pesic, M., Schonenberg, M. H., Sidorova, N. and van der Aalst, W. M. P.: 2007, Constraint-based workflow models: Change made easy, *OTM Conferences (1)*, pp. 77–94.

Petri, C. A.: 1962, *Kommunikation mit Automaten*, PhD thesis, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Darmstadt, Germany.

Pistore, M., Marconi, A., Bertoli, P. and Traverso, P.: 2005, Automated Composition of Web Services by Planning at the Knowledge Level, *19th Int. Joint Conference on Artificial Intelligence*, pp. 1252–1259.

Pnueli, A.: 1977, The temporal logic of programs, *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46 –57.

Pohl, K., Böckle, G. and Linden, F. J. v. d.: 2005, *Software Product Line Engineering*, Springer.

Reichert, M. and Dadam, P.: 1998, ADEPTflex - supporting dynamic changes of workflows without loosing control, *Journal of Intelligent Information Systems* **10**, 93–129.

Reiter, R.: 2001, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press.

Rinderle, S., Reichert, M. and Dadam, P.: 2004, Correctness criteria for dynamic changes in workflow systems a survey, *Data and Knowledge Engineering* **50**, 9–34.

Rodríguez-Moreno, M. D., Borrajo, D., Cesta, A. and Oddi, A.: 2007, Integrating planning and scheduling in workflow domains, *Expert Systems and Applications* **33**(2), 389–406.

Rodríguez-Moreno, M. D. and Kearney, P.: 2002, Integrating ai planning techniques with workflow management system, *Knowledge-Based Systems* **15**(5-6), 285–291.

Sadiq, S. W., Orlowska, M. E. and Sadiq, W.: 2005, Specification and validation of process constraints for flexible workflows, *Information Systems* **30**(5), 349–378.

Sarshar, K. and Loos, P.: 2005, Comparing the control-flow of EPC and petri net from the end-user perspective, *Business Process Management*, Vol. 3649, pp. 434–439.

Schnieders, A. and Puhlmann, F.: 2007, Variability modeling and product derivation in e-business process families, *Technologies for Business Information Systems*, pp. 63–74.

Schonenberg, H., Mans, R., Russell, N., Mulyar, N. and van der Aalst, W. M. P.: 2008, Process flexibility: A survey of contemporary approaches, *CIAO! / EOMAS*, Vol. 10 of *LNBIP*, Springer, pp. 16–30.

Sinnema, M., Deelstra, S., Nijhuis, J. and Bosch, J.: 2006, Modeling dependencies in product families with covamof, *IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS '06)*, IEEE Computer Society, pp. 299–307.

Sohrabi, S. and McIlraith, S. A.: 2010, Preference-based web service composition: A middle ground between execution and search, *Proc. of 9th Int. Semantic Web Conf. (ISWC)*, pp. 713–729.

Sun, C. and Aiello, M.: 2008, Towards variable service compositions using VxBPEL, *ICSR*, pp. 257–261.

Sun, S., Kumar, A. and Yen, J.: 2006, Merging workflows: A new perspective on connecting business processes, *Decision Support Systems* **42**(2), 844–858.

Urban, S., Gao, L., Shrestha, R. and Courter, A.: 2011, The dynamics of process modeling: New directions for the use of events and rules in service-oriented computing, *The Evolution of Conceptual Modeling*, Vol. 6520 of *LNCS*, pp. 205–224.

van Beest, N., Bulanov, P., Wortmann, J. and Lazovik, A.: 2010, Resolving business process interference via dynamic reconfiguration, *8th International Conference on Service Oriented Computing (ICSOC-2010)*, Vol. 6470/2010, Lecture Notes in Computer Science, pp. 47–60.

van Beest, N., Kaldeli, E., Bulanov, P., Wortmann, J. and Lazovik, A.: 2012, Automated runtime repair of business processes, *Technical report*, University of Groningen. http://www.cs.rug.nl/˜eirini/papers/tech_2012-12-2.pdf.

van Beest, N. R. T. P., Szirbik, N. B. and Wortmann, J. C.: 2010, Assessing the interference in concurrent business processes, *Proc. of 12th Int. Conf. on Enterprise Information Systems (ICEIS)*, pp. 261–270.

van der Aalst, W.: 2003, Patterns and xpdl: A critical evaluation of the xml process definition language.

van der Aalst, W. and Dongen, B. F. V.: 2002, Discovering workflow performance models from timed logs, *International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002), volume 2480 of Lecture Notes in Computer Science*, pp. 45–63.

van der Aalst, W. M. P. and Basten, T.: 2002, Inheritance of workflows: an approach to tackling problems related to change, *Theor. Comput. Sci.* **270**(1-2), 125–203.

van der Aalst, W. M. P., van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G. and Weijters, A. J. M. M.: 2003, Workflow mining: A survey of issues and approaches, *Data Knowl. Eng.* **47**(2), 237–267.

van der Aalst, W. M. P., Weske, M. and Grünbauer, D.: 2005, Case handling: a new paradigm for business process support, *Data Knowl. Eng.* **53**(2), 129–162.

van der Aalst, W. and Pesic, M.: 2006, DecSerFlow: Towards a truly declarative service flow language, *Web Services and Formal Methods*, Vol. 4184, pp. 1–23.

van der Aalst, W., ter Hofstede, A. H. M. and Weske, M.: 2003, Business process management: A survey, *Business Process Management*, pp. 1–12.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B. and Barros, A.: 2003, Workflow patterns, *Distributed and Parallel Databases* **14(3)**, 5–51.

Vanhatalo, J., Völzer, H. and Leymann, F.: 2007, Faster and more focused control-flow analysis for business process models through sese decomposition, *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, pp. 43–55.

Vilain, M. B. and Kautz, H. A.: 1986, Constraint propagation algorithms for temporal reasoning, *AAAI*, pp. 377–382.

Weber, B., Reichert, M. and Rinderle-Ma, S.: 2008, Change patterns and change support features - enhancing flexibility in process-aware information systems, *Data Knowl. Eng.* **66**(3), 438–466.

World Wide Web Consorcium (W3C): 2004, OWL-S: semantic markup for web services, *Technical Report SUBM-OWL-S-20041122*, W3C. http://www.w3.org/Submission/OWL-S, accessible July 2012.

World Wide Web Consorcium (W3C): 2005, Web Service Semantics – WSDL-S, *Technical Report SUBM-WSDL-S-20051107*, W3C. http://www.w3.org/Submission/WSDL-S, accessible July 2012.

Wynn, M., Verbeek, H., van der Aalst, W., ter Hofstede, A. and Edmond, D.: 2009, Business process verification : finally a reality!, *Business Process Management Journal* **15**(1), 74–92.

Xiao, Y. and Urban, S.: 2008, Using data dependencies to support the recovery of concurrent processes in a service composition environment, *Proc. of the 16th Int. Conf. on Cooperative Inf. Systems*.